PGI® User's Guide

Parallel Fortran, C and C++ for Scientists and Engineers

The Portland Group®
STMicroelectronics
Two Centerpointe Drive
Lake Oswego, OR 97035

While every precaution has been taken in the preparation of this document, The Portland Group® (PGI®), a wholly-owned subsidiary of STMicroelectronics, Inc., makes no warranty for the use of its products and assumes no responsibility for any errors that may appear, or for damages resulting from the use of the information contained herein. The Portland Group retains the right to make changes to this information at any time, without notice. The software described in this document is distributed under license from STMicroelectronics and/or The Portland Group and may be used or copied only in accordance with the terms of the license agreement ("EULA").

No part of this document may be reproduced or transmitted in any form or by any means, for any purpose other than the purchaser's or the end user's personal use without the express written permission of STMicroelectronics and/or The Portland Group.

PGI® User's Guide

Copyright © 1998 – 2000 The Portland Group, Inc.
Copyright © 2000 – 2009 STMicroelectronics, Inc.
All rights reserved.

Printed in the United States of America
First Printing: Release 1.7, Jun 1998
Second Printing: Release 3.0, Jan 1999
Third Printing: Release 3.1, Sep 1999
Fourth Printing: Release 3.2, Sep 2000
Fifth Printing: Release 4.0, May 2002
Sixth Printing: Release 5.0, Jun 2003
Seventh Printing: Release 5.1, Nov 2003
Eight Printing: Release 5.2, Jun 2004
Ninth Printing: Release 6.0, Mar 2005
Tenth Printing: Release 6.1, Dec 2005
Eleventh Printing: Release 6.2, August 2006
Twelfth printing: Release 7.0, December, 2006
Thirteenth printing: Release 7.1, October, 2007
Fourteenth printing: Release 7.2, May, 2008
Fourteenth printing: Release 8.0, November, 2008
Fifteenth printing: Release 9.0, June, 2009

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
Web: www.pgroup.com

ID: 091661239

Contents

Preface xxi
Audience Description xxi
Compatibility and Conformance to Standards xxi
Organization xxii
Hardware and Software Constraints xxiv
Conventions xxiv
Related Publications xxvi

1. Getting Started 1
Overview 1
Invoking the Command-level PGI Compilers 1
Command-line Syntax 2
Command-line Options 3
Fortran Directives and C/C++ Pragmas 3
Filename Conventions 3
Input Files 4
Output Files 5
Fortran, C, and C++ Data Types 7
Parallel Programming Using the PGI Compilers 7
Running SMP Parallel Programs 8
Running Data Parallel HPF Programs 8
Platform-specific considerations 8
Using the PGI Compilers on Linux 9
Using the PGI Compilers on Windows 10
Using the PGI Compilers on SUA and SFU 11
Using the PGI Compilers on Mac OS X 12
Site-specific Customization of the Compilers 13
Using siterc Files 13
Using User rc Files 13
Common Development Tasks 14

2. Using Command Line Options 17
Command Line Option Overview 17

Command-line Options Syntax 17
Command-line Suboptions 18
Command-line Conflicting Options 18
Help with Command-line Options 18
Getting Started with Performance 19
Using –fast and –fastsse Options 20
Other Performance-related Options 20
Targeting Multiple Systems - Using the -tp Option 21
Frequently-used Options 21

3. Optimizing & Parallelizing 23
Overview of Optimization 24
Local Optimization 24
Global Optimization 24
Loop Optimization: Unrolling, Vectorization, and Parallelization 24
Interprocedural Analysis (IPA) and Optimization 24
Function Inlining 24
Profile-Feedback Optimization (PFO) 25
Getting Started with Optimizations 25
Common Compiler Feedback Format (CCFF) 27
Local and Global Optimization using -O 27
Scalar SSE Code Generation 28
Loop Unrolling using –Munroll 29
Vectorization using –Mvect 30
Vectorization Sub-options 30
Vectorization Example Using SSE/SSE2 Instructions 32
Auto-Parallelization using -Mconcur 35
Auto-parallelization Sub-options 35
Loops That Fail to Parallelize 36
Processor-Specific Optimization & the Unified Binary 39
Interprocedural Analysis and Optimization using –Mipa 39
Building a Program Without IPA – Single Step 39
Building a Program Without IPA - Several Steps 40
Building a Program Without IPA Using Make 40
Building a Program with IPA 41
Building a Program with IPA - Single Step 41
Building a Program with IPA - Several Steps 42
Building a Program with IPA Using Make 42
Questions about IPA 43
Profile-Feedback Optimization using –Mpfi/–Mpfo 44
Default Optimization Levels 44
Local Optimization Using Directives and Pragmas 45
Execution Timing and Instruction Counting 45
Portability of Multi-Threaded Programs on Linux 46
libpgbind 46
libnuma 47

4. Using Function Inlining 49
Invoking Function Inlining 49
Using an Inline Library 50
Creating an Inline Library 51
Working with Inline Libraries 52
Updating Inline Libraries - Makefiles 52
Error Detection during Inlining 53
Examples 53
Restrictions on Inlining 53

5. Using OpenMP 55
OpenMP Overview 55
OpenMP Shared-Memory Parallel Programming Model 55
Terminology 56
OpenMP Example 57
Task Overview 58
Fortran Parallelization Directives 59
C/C++ Parallelization Pragmas 59
Directive and Pragma Recognition 60
Directive and Pragma Summary Table 60
Directive and Pragma Clauses 62
Run-time Library Routines 65
Environment Variables 70

6. Using MPI 73
MPI Overview 73
Compiling and Linking MPI Applications 74
Debugging MPI Applications 74
Profiling MPI Applications 75
Using MPICH-1 on Linux 75
Using MPICH-2 on Linux 76
Using MVAPICH on Linux 76
Using HP-MPI on Linux 77
Using OpenMPI on Linux 77
Compiling using OpenMPI 77
Generate MPI Profile Data 78
Using MSMPI on Windows 78
MSMPI Environment 78
Compiling using MSMPI 78
Generate MPI Profile Data 78
Using OpenMPI on Mac OS X 79
Compiling using OpenMPI 79
Generate MPI Profile Data 79

7. Using an Accelerator 81
Overview 81

Components 81
Availability 81
User-directed Accelerator Programming 82
Features Not Covered or Implemented 82
Terminology 82
System Requirements 83
Supported Processors and GPUs 83
Installation and Licensing 84
Required Files 84
Command Line Flag 84
Execution Model 85
Host Functions 85
Levels of Parallelism 85
Memory Model 86
Separate Host and Accelerator Memory Considerations 86
Accelerator Memory 86
Cache Management 86
Running an Accelerator Program 87
Directives 87
Enable Accelerator Directives 87
Format 87
C Directives 88
Free-Form Fortran Directives 88
Fixed-Form Fortran Directives 89
Accelerator Directive Summary 90
Accelerator Directive Clauses 91
PGI Accelerator Compilers Runtime Libraries 92
Runtime Library Definitions 92
Runtime Library Routines 93
Environment Variables 93
Applicable Command Line Options 94
PGI Unified Binary for Accelerators 95
Multiple Processor Targets 96
Profiling Accelerator Kernels 96
Related Accelerator Programming Tools 97
NVIDIA CUDA Profile 97
TAU - Tuning and Analysis Utility 97
Supported Intrinsics 97
Supported Fortran Intrinsics Summary Table 97
Supported C Intrinsics Summary Table 99
References related to Accelerators 100

8. Using Directives and Pragmas 101
PGI Proprietary Fortran Directives 101
PGI Proprietary C and C++ Pragmas 102
PGI Proprietary Optimization Directive and Pragma Summary 102

Scope of Fortran Directives and Command-Line options 104
Scope of C/C++ Pragmas and Command-Line Options 105
Prefetch Directives and Pragmas 107
Prefetch Directive Syntax 108
Prefetch Directive Format Requirements 108
Sample Usage of Prefetch Directive 108
Prefetch Pragma Syntax 109
Sample Usage of Prefetch Pragma 109
!DEC$ Directives 109
Format Requirements 109
Summary Table 109
C$PRAGMA C 110

9. Creating and Using Libraries 111
Using builtin Math Functions in C/C++ 111
Using System Library Routines 112
Creating and Using Shared Object Files on Linux 112
Creating and Using Shared Object Files in SFU and 32-bit SUA 114
Shared Object Error Message 114
Shared Object-Related Compiler Switches 114
Creating and Using Dynamic Libraries on Mac OS X 115
PGI Runtime Libraries on Windows 116
Creating and Using Static Libraries on Windows 116
ar command 116
ranlib command 117
Creating and Using Dynamic-Link Libraries on Windows 118
Using LIB3F 125
LAPACK, BLAS and FFTs 125
The C++ Standard Template Library 125

10. Using Environment Variables 127
Setting Environment Variables 127
Setting Environment Variables on Linux 127
Setting Environment Variables on Windows 128
Setting Environment Variables on Mac OSX 128
PGI-Related Environment Variables 129
PGI Environment Variables 130
FLEXLM_BATCH 131
FORTRANOPT 131
GMON_OUT_PREFIX 131
LD_LIBRARY_PATH 131
LM_LICENSE_FILE 132
MANPATH 132
MPSTKZ 132
MP_BIND 132
MP_BLIST 133

MP_SPIN 133
MP_WARN 133
NCPUS 133
NCPUS_MAX 134
NO_STOP_MESSAGE 134
PATH 134
PGI 134
PGI_CONTINUE 135
PGI_OBJSUFFIX 135
PGI_STACK_USAGE 135
PGI_TERM 135
PGI_TERM_DEBUG 137
PWD 137
STATIC_RANDOM_SEED 137
TMP 138
TMPDIR 138
Using Environment Modules on Linux 138
Stack Traceback and JIT Debugging 139

11. Distributing Files - Deployment 141
Deploying Applications on Linux 141
Runtime Library Considerations 141
64-bit Linux Considerations 142
Linux Redistributable Files 142
Restrictions on Linux Portability 142
Installing the Linux Portability Package 142
Licensing for Redistributable Files 143
Deploying Applications on Windows 143
PGI Redistributables 143
Microsoft Redistributables 143
Code Generation and Processor Architecture 144
Generating Generic x86 Code 144
Generating Code for a Specific Processor 144
Generating One Executable for Multiple Types of Processors 144
PGI Unified Binary Command-line Switches 145
PGI Unified Binary Directives and Pragmas 145

12. Inter-language Calling 147
Overview of Calling Conventions 147
Inter-language Calling Considerations 148
Functions and Subroutines 148
Upper and Lower Case Conventions, Underscores 149
Compatible Data Types 149
Fortran Named Common Blocks 150
Argument Passing and Return Values 151
Passing by Value (%VAL) 151

Character Return Values 151
Complex Return Values 152
Array Indices 152
Examples 153
Example - Fortran Calling C 153
Example - C Calling Fortran 154
Example - C++ Calling C 155
Example - C Calling C++ 155
Example - Fortran Calling C++ 156
Example - C++ Calling Fortran 157
Win32 Calling Conventions 158
Win32 Fortran Calling Conventions 158
Symbol Name Construction and Calling Example 159
Using the Default Calling Convention 160
Using the STDCALL Calling Convention 160
Using the C Calling Convention 161
Using the UNIX Calling Convention 161
Using the CREF Calling Convention 161

13. Programming Considerations for 64-Bit Environments 163
Data Types in the 64-Bit Environment 163
C/C++ Data Types 164
Fortran Data Types 164
Large Static Data in Linux 164
Large Dynamically Allocated Data 164
64-Bit Array Indexing 164
Compiler Options for 64-bit Programming 165
Practical Limitations of Large Array Programming 166
Medium Memory Model and Large Array in C 167
Medium Memory Model and Large Array in Fortran 168
Large Array and Small Memory Model in Fortran 169

14. C/C++ Inline Assembly and Intrinsics 171
Inline Assembly 171
Extended Inline Assembly 172
Output Operands 173
Input Operands 175
Clobber List 176
Additional Constraints 177
Operand Aliases 183
Assembly String Modifiers 183
Extended Asm Macros 185
Intrinsics 186

15. Fortran, C, and C++ Data Types 187
Fortran Data Types 187

Fortran Scalars 187
FORTRAN 77 Aggregate Data Type Extensions 189
Fortran 90 Aggregate Data Types (Derived Types) 190
C and C++ Data Types 190
C and C++ Scalars 190
C and C++ Aggregate Data Types 192
Class and Object Data Layout 192
Aggregate Alignment 193
Bit-field Alignment 194
Other Type Keywords in C and C++ 194

16. Command-Line Options Reference 195
PGI Compiler Option Summary 195
Build-Related PGI Options 195
PGI Debug-Related Compiler Options 198
PGI Optimization-Related Compiler Options 199
PGI Linking and Runtime-Related Compiler Options 199
C and C++ Compiler Options 200
Generic PGI Compiler Options 203
C and C++ -specific Compiler Options 242
–M Options by Category 253
Code Generation Controls 254
C/C++ Language Controls 258
Environment Controls 259
Fortran Language Controls 261
Inlining Controls 263
Optimization Controls 264
Miscellaneous Controls 275

17. OpenMP Reference Information 283
Tasks 283
Task Characteristics and Activities 283
Task Scheduling Points 283
Task Construct 284
Parallelization Directives and Pragmas 286
ATOMIC and atomic 286
BARRIER and barrier 287
CRITICAL ... END CRITICAL and critical 287
C$DOACROSS 289
DO...END DO and for 290
FLUSH and flush 292
MASTER ... END MASTER and master 292
ORDERED and ordered 293
PARALLEL ... END PARALLEL and parallel 294
PARALLEL DO 296
PARALLEL SECTIONS and parallel sections 296

PARALLEL WORKSHARE ... *END PARALLEL WORKSHARE* 297
SECTIONS ... END SECTIONS and sections 298
SINGLE ... END SINGLE and single 299
TASK and task 300
TASKWAIT and taskwait 301
THREADPRIVATE and threadprivate 302
WORKSHARE ... END WORKSHARE 303
Directive and Pragma Clauses 303
COLLAPSE (n) 304
COPYIN (list) 304
COPYPRIVATE(list) 304
DEFAULT 305
FIRSTPRIVATE(list) 305
IF() 305
LASTPRIVATE(list) 305
NOWAIT 305
NUM_THREADS 306
ORDERED 306
PRIVATE 306
REDUCTION 306
SCHEDULE 307
SHARED 308
UNTIED 308
OpenMP Environment Variables 308
OMP_DYNAMIC 308
OMP_NESTED 308
OMP_MAX_ACTIVE_LEVELS 308
OMP_NUM_THREADS 309
OMP_SCHEDULE 309
OMP_STACKSIZE 309
OMP_THREAD_LIMIT 310
OMP_WAIT_POLICY 310

18. PGI Accelerator Compilers Reference 311
PGI Accelerator Directives 311
Accelerator Region Directive 312
Accelerator Loop Mapping Directive 313
Combined Directives 313
PGI Accelerator Directive Clauses 314
if clause 314
Data Clauses 314
copy clause 316
copyin clause 316
copyout clause 316
local clause 316
loop Directive Clauses 316

host clause 317
parallel clause 317
seq clause 318
vector clause 318
kernel clause 318
private clause 318
PGI Accelerator Runtime Routines 319
acc_get_device 319
acc_get_num_devices 319
acc_init 320
acc_set_device_num 320
Environment Variables 321
ACC_DEVICE 321
ACC_DEVICE_NUM 321
ACC_NOTIFY 322

19. C++ Name Mangling 323
Types of Mangling 324
Mangling Summary 324
Type Name Mangling 324
Nested Class Name Mangling 325
Local Class Name Mangling 325
Template Class Name Mangling 325

20. Directives and Pragmas Reference 327
PGI Proprietary Fortran Directive and C/C++ Pragma Summary 327
altcode (noaltcode) 328
assoc (noassoc) 329
bounds (nobounds) 329
cncall (nocncall) 329
concur (noconcur) 329
depchk (nodepchk) 329
eqvchk (noeqvchk) 329
fcon (nofcon) 330
invarif (noinvarif) 330
ivdep 330
lstval (nolstval) 330
prefetch 330
opt 330
safe (nosafe) 331
safe_lastval 331
safeptr (nosafeptr) 332
single (nosingle) 333
tp 333
unroll (nounroll) 333
vector (novector) 334

vintr (novintr) 334
Prefetch Directives and Pragmas 334
!DEC$ Directives 334
ALIAS Directive 334
ATTRIBUTES Directive 335
DECORATE Directive 336
DISTRIBUTE Directive 336
IGNORE_TKR Directive 336

21. Run-time Environment 339
Linux86 and Win32 Programming Model 339
Function Calling Sequence 339
Function Return Values 342
Argument Passing 343
Linux86-64 Programming Model 345
Function Calling Sequence 346
Function Return Values 348
Argument Passing 349
Linux86-64 Fortran Supplement 351
Win64 Programming Model 356
Function Calling Sequence 356
Function Return Values 358
Argument Passing 359
Win64/SUA64 Fortran Supplement 361

22. C++ Dialect Supported 367
Extensions Accepted in Normal C++ Mode 367
cfront 2.1 Compatibility Mode 368
cfront 2.1/3.0 Compatibility Mode 370

23. Fortran Module/Library Interfaces for Windows 371
Source Files 371
Data Types 371
Using DFLIB and DFPORT 372
DFLIB 372
DFPORT 373
Using the DFWIN module 379
Supported Libraries and Modules 379
advapi32 379
comdlg32 381
dfwbase 382
dfwinty 382
gdi32 382
kernel32 385
shell32 393
user32 394

winver 398
wsock32 398

24. C/C++ MMX/SSE Inline Intrinsics 401
Using Intrinsic functions 401
Required Header File 401
Intrinsic Data Types 402
Intrinsic Example 402
MMX Intrinsics 402
SSE Intrinsics 404
ABM Intrinsics 408

25. Messages 409
Diagnostic Messages 409
Phase Invocation Messages 410
Fortran Compiler Error Messages 410
Message Format 410
Message List 410
Fortran Run-time Error Messages 435
Message Format 435
Message List 435

Index 439

Figures

15.1. Internal Padding in a Structure 193
15.2. Tail Padding in a Structure 194

Tables

1. PGI Compilers and Commands xxv
1.1. Stop-after Options, Inputs and Outputs 6
1.2. Examples of Using siterc and User rc Files 13
2.1. Commonly Used Command Line Options 21
3.1. Optimization and –O, –g and –M<opt> Options 44
5.1. Directive and Pragma Summary Table 61
5.2. Directive and Pragma Clauses Summary Table 62
5.3. Run-time Library Routines Summary 65
5.4. OpenMP-related Environment Variable Summary Table 71
6.1. MPI Implementation Options 74
6.2. MPI Profiling Options 75
7.1. PGI Accelerator Directive Summary Table 90
7.2. Region Clauses Summary 91
7.3. Loop Clauses Summary 92
7.4. Accelerator Runtime Library Routines 93
7.5. Accelerator Environment Variables Routines 93
7.6. Supported Fortran Intrinsic Summary Table 98
7.7. Supported C Intrinsic Double Functions 99
7.8. Supported C Intrinsic Float Functions 99
8.1. Proprietary Optimization-Related Fortran Directive and C/C++ Pragma Summary 103
8.2. !DEC$ Directives Summary Table 110
10.1. PGI-Related Environment Variable Summary 129
10.2. Supported PGI_TERM Values 136
12.1. Fortran and C/C++ Data Type Compatibility 149
12.2. Fortran and C/C++ Representation of the COMPLEX Type 150
12.3. Calling Conventions Supported by the PGI Fortran Compilers 159
13.1. 64-bit Compiler Options 165
13.2. Effects of Options on Memory and Array Sizes 166
13.3. 64-Bit Limitations 166
14.1. Simple Constraints 178
14.2. x86/x86_64 Machine Constraints 179
14.3. Multiple Alternative Constraints 181
14.4. Constraint Modifier Characters 182

14.5. Assembly String Modifier Characters 184
14.6. Intrinsic Header File Organization 186
15.1. Representation of Fortran Data Types 187
15.2. Real Data Type Ranges 188
15.3. Scalar Type Alignment 188
15.4. C/C++ Scalar Data Types 190
15.5. Scalar Alignment 191
16.1. PGI Build-Related Compiler Options 196
16.2. PGI Debug-Related Compiler Options 198
16.3. Optimization-Related PGI Compiler Options 199
16.4. Linking and Runtime-Related PGI Compiler Options 199
16.5. C and C++ -specific Compiler Options 200
16.6. Subgroups for –help Option 213
16.7. –M Options Summary 218
16.8. Optimization and –O, –g, –Mvect, and –Mconcur Options 226
17.1. Initialization of REDUCTION Variables 306
20.1. IGNORE_TKR Example 337
21.1. Register Allocation 340
21.2. Standard Stack Frame 340
21.3. Stack Contents for Functions Returning struct/union 343
21.4. Integral and Pointer Arguments 343
21.5. Floating-point Arguments 343
21.6. Structure and Union Arguments 344
21.7. Register Allocation 346
21.8. Standard Stack Frame 346
21.9. Register Allocation for Example A-2 350
21.10. Linux86-64 Fortran Fundamental Types 352
21.11. Fortran and C/C++ Data Type Compatibility 353
21.12. Fortran and C/C++ Representation of the COMPLEX Type 354
21.13. Register Allocation 356
21.14. Standard Stack Frame 357
21.15. Register Allocation for Example A-4 360
21.16. Win64 Fortran Fundamental Types 361
21.17. Fortran and C/C++ Data Type Compatibility 363
21.18. Fortran and C/C++ Representation of the COMPLEX Type 364
23.1. Fortran Data Type Mappings 371
23.2. DFLIB Function Summary 372
23.3. DFPORT Functions 373
23.4. DFWIN advapi32 Functions 380
24.1. MMX Intrinsics (mmintrin.h) 403
24.2. SSE Intrinsics (xmmintrin.h) 404
24.3. SSE2 Intrinsics (emmintrin.h) 405
24.4. SSE3 Intrinsics (pmmintrin.h) 407
24.5. SSSE3 Intrinsics (tmmintrin.h) 407
24.6. SSE4a Intrinsics (ammintrin.h) 408
24.7. ABM Intrinsics (intrin.h) 408

Examples

1.1. Hello program 2
2.1. Makefiles with Options 18
3.1. Dot Product Code 30
3.2. Unrolled Dot Product Code 30
3.3. Vector operation using SSE instructions 33
3.4. Using SYSTEM_CLOCK code fragment 46
4.1. Sample Makefile 52
5.1. OpenMP Loop Example 57
6.1. MPI Hello World Example 76
7.1. Accelerator Kernel Timing Data 96
8.1. Prefetch Directive Use 108
8.2. Prefetch Pragma in C 109
9.1. Build a DLL: Fortran 119
9.2. Build a DLL: C 120
9.3. Build DLLs Containing Circular Mutual Imports: C 121
9.4. Build DLLs Containing Mutual Imports: Fortran 122
9.5. Import a Fortran module from a DLL 124
12.1. Character Return Parameters 152
12.2. COMPLEX Return Values 152
12.3. Fortran Main Program f2c_main.f 153
12.4. C function f2c_func_ 153
12.5. C Main Program c2f_main.c 154
12.6. Fortran Subroutine c2f_sub.f 154
12.7. C++ Main Program cp2c_main.C Calling a C Function 155
12.8. Simple C Function c2cp_func.c 155
12.9. C Main Program c2cp_main.c Calling a C++ Function 155
12.10. Simple C++ Function c2cp_func.C with Extern C 156
12.11. Fortran Main Program f2cp_main.f calling a C++ function 156
12.12. C++ function f2cp_func.C 156
12.13. C++ main program cp2f_main.C 157
12.14. Fortran Subroutine cp2f_func.f 157
13.1. Medium Memory Model and Large Array in C 167
13.2. Medium Memory Model and Large Array in Fortran 168

13.3. Large Array and Small Memory Model in Fortran 169
17.1. OpenMP Task C Example 285
17.2. OpenMP Task Fortran Example 285
21.1. C Program Calling an Assembly-language Routine 345
21.2. Parameter Passing 350
21.3. C Program Calling an Assembly-language Routine 351
21.4. Parameter Passing 359
21.5. C Program Calling an Assembly-language Routine 360

Preface

This guide is part of a set of manuals that describe how to use The Portland Group (PGI) Fortran, C, and C++ compilers and program development tools. These compilers and tools include the *PGF77*, *PGF95*, *PGHPF*, *PGC++*, and *PGCC ANSI* C compilers, the *PGPROF* profiler, and the *PGDBG* debugger. They work in conjunction with an x86 or x64 assembler and linker. You can use the PGI compilers and tools to compile, debug, optimize, and profile serial and parallel applications for x86 (Intel Pentium II/III/4/M, Intel Centrino, Intel Xeon, AMD Athlon XP/MP) or x64 (AMD Athlon64/Opteron/Turion, Intel EM64T, Intel Core Duo, Intel Core 2 Duo, Barcelona) processor-based systems.

The *PGI User's Guide* provides operating instructions for the PGI command-level development environment. It also contains details concerning the PGI compilers' interpretation of the Fortran language, implementation of Fortran language extensions, and command-level compilation. Users are expected to have previous experience with or knowledge of the Fortran programming language.

Audience Description

This manual is intended for scientists and engineers using the PGI compilers. To use these compilers, you should be aware of the role of high-level languages, such as Fortran, C, and C++, as well as assembly-language in the software development process; and you should have some level of understanding of programming. The PGI compilers are available on a variety of x86 or x64 hardware platforms and operating systems. You need to be familiar with the basic commands available on your system.

Compatibility and Conformance to Standards

Your system needs to be running a properly installed and configured version of the compilers. For information on installing PGI compilers and tools, refer to the Release Notes and Installation Guide included with your software.

For further information, refer to the following:

- *American National Standard Programming Language FORTRAN*, ANSI X3. -1978 (1978).
- *ISO/IEC 1539-1 : 1991, Information technology – Programming Languages – Fortran,* Geneva, 1991 (Fortran 90).
- *ISO/IEC 1539-1 : 1997, Information technology – Programming Languages – Fortran,* Geneva, 1997 (Fortran 95).

- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *High Performance Fortran Language Specification*, Revision 1.0, Rice University, Houston, Texas (1993), http://www.crpc.rice.edu/HPFF.
- *High Performance Fortran Language Specification*, Revision 2.0, Rice University, Houston, Texas (1997), http://www.crpc.rice.edu/HPFF.
- *OpenMP Application Program Interface*, Version 2.5, May 2005, http://www.openmp.org.
- *Programming in VAX Fortran*, Version 4.0, Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).
- *American National Standard Programming Language C*, ANSI X3.159-1989.
- ISO/IEC 9899:1999, Information technology – Programming Languages – C, Geneva, 1999 (C99).

Organization

Users typically begin by wanting to know how to use a product and often then find that they need more information and facts about specific areas of the product. Knowing *how* as well as *why* you might use certain options or perform certain tasks is key to using the PGI compilers and tools effectively and efficiently. However, once you have this knowledge and understanding, you very likely might find yourself wanting to know much more about specific areas or specific topics.

To facilitate ease of use, this manual is divided into the following two parts:

- Part I, Compiler Usage, contains the essential information on how to use the compiler.
- Part II, Reference Information, contains more detailed reference information about specific aspects of the compiler, such as the details of compiler options, directives, and more.

Part I, Compiler Usage, contains these chapters:

Chapter 1, "*Getting Started*" provides an introduction to the PGI compilers and describes their use and overall features.

Chapter 2, "*Using Command Line Options*" provides an overview of the command-line options as well as task-related lists of options.

Chapter 3, "*Optimizing & Parallelizing*" describes standard optimization techniques that, with little effort, allow users to significantly improve the performance of programs.

Chapter 4, "*Using Function Inlining*" describes how to use function inlining and shows how to create an inline library.

Chapter 5, "*Using OpenMP*" provides a description of the OpenMP Fortran parallelization directives and of the OpenMP C and C++ parallelization pragmas and shows examples of their use.

Chapter 6, "*Using MPI* " describes how to use MPI with PGI Workstation and PGI server.

Chapter 7, "*Using an Accelerator*" describes how to use thePGI Accelerator compilers.

Chapter 8, "*Using Directives and Pragmas*" provides a description of each Fortran optimization directive and C/C++ optimization pragma, and shows examples of their use.

Chapter 9, "*Creating and Using Libraries*" discusses PGI support libraries, shared object files, and environment variables that affect the behavior of the PGI compilers.

Chapter 10, " *Using Environment Variables*" describes the environment variables that affect the behavior of the PGI compilers.

Chapter 11, "*Distributing Files - Deployment*" describes the deployment of your files once you have built, debugged and compiled them successfully.

Chapter 12, "*Inter-language Calling*" provides examples showing how to place C Language calls in a Fortran program and Fortran Language calls in a C program.

Chapter 13, "*Programming Considerations for 64-Bit Environments*" discusses issues of which programmers should be aware when targeting 64-bit processors.

Chapter 14, "*C/C++ Inline Assembly and Intrinsics*" describes how to use inline assembly code in C and C++ programs, as well as how to use intrinsic functions that map directly to x86 and x64 machine instructions.

Part II, Reference Information, contains these chapters:

Chapter 15, "*Fortran, C, and C++ Data Types*" describes the data types that are supported by the PGI Fortran, C, and C++ compilers.

Chapter 16, "*Command-Line Options Reference*" provides a detailed description of each command-line option.

Chapter 17, "*OpenMP Reference Information*"contains detailed descriptions of each of the OpenMP directives and pragmas that PGI supports.

Chapter 18, "*PGI Accelerator Compilers Reference* "contains detailed descriptions of each of the PGI Accelerator directives, runtime routines, and environment variables that PGI supports.

Chapter 19, "*C++ Name Mangling*" describes the name mangling facility and explains the transformations of names of entities to names that include information on aspects of the entity's type and a fully qualified name.

Chapter 20, "*Directives and Pragmas Reference*"contains detailed descriptions of PGI's proprietary directives and pragmas.

Chapter 21, "*Run-time Environment*" describes the assembly language calling conventions and examples of assembly language calls.

Chapter 22, "*C++ Dialect Supported*" lists more details of the version of the C++ language that PGC++ supports.

Chapter 23, "*Fortran Module/Library Interfaces for Windows*" provides a description of the Fortran module library interfaces that PVF supports, describing each property available.

Chapter 24, "*C/C++ MMX/SSE Inline Intrinsics*" provides tables that list the MMX Inline Intrinsics (mmintrin.h), the SSE1 inline intrinsics (xmmintrin.h), and SSE2 inline intrinsics (emmintrin.h).

Chapter 25, "*Messages*" provides a list of compiler error messages.

Hardware and Software Constraints

This guide describes versions of the PGI compilers that produce assembly code for x86 and x64 processor-based systems. Details concerning environment-specific values and defaults and system-specific features or limitations are presented in the release notes delivered with the PGI compilers.

Conventions

The *PGI User's Guide* uses the following conventions:

italic
: Italic font is for emphasis.

`Constant Width`
: Constant width font is for commands, filenames, directories, examples and for language statements in the text, including assembly language statements.

[item1]
: Square brackets indicate optional items. In this case item1 is optional.

{ item2 | item 3}
: Braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename...
: Ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN
: Fortran language statements are shown in the text of this guide using upper-case characters and a reduced point size.

The PGI compilers and tools are supported on both 32-bit and 64-bit variants of Linux, Mac OS X, and Windows operating systems on a variety of x86-compatible processors. There are a wide variety of releases and distributions of each of these types of operating systems. Further, The *PGI User's Guide* uses a number of terms with respect to these platforms. For a complete definition of these terms and other terms in this guide with which you may be unfamiliar, PGI provides a glossary of terms which you can access at *www.pgroup.com/support/definitions.htm*.

AMD64	linux86	osx86	SSSE3
barcelona	linux86-64	osx86-64	static linking
DLL	Mac OS X	SFU	SUA
driver	-mcmodel=small	shared library	Win32
dynamic library	-mcmodel=medium	SSE	Win64

EM64T	MPI	SSE1	Windows
hyperthreading (HT)	MPICH	SSE2	x64
IA32	multi-core	SSE3	x86
Large arrays	NUMA	SSE4A and ABM	x87

The following table lists the PGI compilers and tools and their corresponding commands:

Table 1. PGI Compilers and Commands

Compiler or Tool	Language or Function	Command
PGF77	FORTRAN 77	pgf77
PGF95	Fortran 90/95	pgf95
PGHPF	High Performance Fortran	pghpf
PGCC C	*ANSI* C99 and K&R C	pgcc
PGC++	*ANSI* C++ with cfront features	pgcpp on Windows pgCC on Linux
PGDBG	Source code debugger	pgdbg
PGPROF	Performance profiler	pgprof

In general, the designation *PGF95* is used to refer to The Portland Group's Fortran 90/95 compiler, and *pgf95* is used to refer to the command that invokes the compiler. A similar convention is used for each of the PGI compilers and tools.

For simplicity, examples of command-line invocation of the compilers generally reference the *pgf95* command, and most source code examples are written in Fortran. Usage of the *PGF77* compiler, whose features are a subset of *PGF95*, is similar. Usage of *PGHPF, PGC++*, and *PGCC* is consistent with *PGF95* and *PGF77*, but there are command-line options and features of these compilers that do not apply to *PGF95* and *PGF77* and vice versa.

There are a wide variety of x86-compatible processors in use. All are supported by the PGI compilers and tools. Most of these processors are forward-compatible, but not backward-compatible, meaning that code compiled to target a given processor will not necessarily execute correctly on a previous-generation processor.

A table listing the processor options that PGI supports is available in the Release Notes. The table also includes the features utilized by the PGI compilers that distinguish them from a compatibility standpoint.

In this manual, the convention is to use "x86" to specify the group of processors that are "32-bit" but not "64-bit." The convention is to use "x64" to specify the group of processors that are both "32-bit" and "64-bit." x86 processor-based systems can run only 32-bit operating systems. x64 processor-based systems can run either 32-bit or 64-bit operating systems, and can execute all 32-bit x86 binaries in either case. x64 processors have additional registers and 64-bit addressing capabilities that are utilized by the PGI compilers and tools when running on a 64-bit operating system. The prefetch, SSE1, SSE2 and SSE3 processor features further distinguish the various processors. Where such distinctions are important with respect to a given compiler option or feature, it is explicitly noted in this manual.

Note

The default for performing scalar floating-point arithmetic is to use SSE instructions on targets that support SSE1 and SSE2.

Related Publications

The following documents contain additional information related to the x86 and x64 architectures, and the compilers and tools available from The Portland Group.

- *PGI Fortran Reference* manual describes the FORTRAN 77, Fortran 90/95, and HPF statements, data types, input/output format specifiers, and additional reference material related to use of the PGI Fortran compilers.
- *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- *System V Application Binary Interface X86-64 Architecture Processor Supplement*, *www.x86-64.org/abi.pdf*.
- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *Programming in VAX Fortran, Version 4.0,* Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran,* IBM Corporation, Rev. GC26-4119.
- *The C Programming Language* by Kernighan and Ritchie (Prentice Hall).
- *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice Hall, 1987).
- *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, Inc. (Addison-Wesley Publishing Co., 1990).
- *OpenMP Application Program Interface*, Version 2.5 May 2005 (OpenMP Architecture Review Board, 1997-2005).

Part I. Compiler Usage

Users typically begin by wanting to know how to use a product and often then find that they need more information and facts about specific areas of the product. Knowing *how* as well as *why* you might use certain options or perform certain tasks is key to using the PGI compilers and tools effectively and efficiently. In the chapters in this part of the guide you learn how to:

- Get started using the PGI compilers, as described in Chapter 1, "Getting Started" on page 1.
- Use the most common command line options and learn why specific ones are especially beneficial for you to use, as described in Chapter 2, "Using Command Line Options" on page 17.
- Use optimization and parallelization to increase the performance of your program, as described in Chapter 3, "Optimizing & Parallelizing" on page 23.
- Invoke function inlining and create an inline library, as described in Chapter 4, "Using Function Inlining" on page 49.
- Use OpenMP directives, pragmas, run-time libraries, and environment variables, as described in Chapter 5, "Using OpenMP" on page 55.
- Use MPI, including compiling, linking and generating MPI profile data, as described in Chapter 6, "Using MPI" on page 73.
- Using PGI Accelerator compilers, as described in Chapter 7, "Using an Accelerator" on page 81.
- Use PGI directives and pragmas, as described in Chapter 8, "Using Directives and Pragmas" on page 101.
- Create and use libraries, as described in Chapter 9, "Creating and Using Libraries" on page 111.
- Create and use environment variables to control the behavior of PGI software, as described in Chapter 10, "Using Environment Variables" on page 127.
- Distribute files and deploy your applications, as described in Chapter 11, "Distributing Files - Deployment" on page 141.
- Make inter-language calls, as described in Chapter 12, "Inter-language Calling" on page 147.
- Incorporate programming considerations for 64-bit environments, as described in Chapter 13, "Programming Considerations for 64-bit Environments" on page 163.
- Properly use C/C++ inline assembly instructions and intrinsics, as described in Chapter 14, "C/C++ Inline Assembly and Intrinsics" on page 171.

Chapter 1. Getting Started

This chapter describes how to use the PGI compilers. The command used to invoke a compiler, such as the pgf95 command, is called a *compiler driver*. The compiler driver controls the following phases of compilation: preprocessing, compiling, assembling, and linking. Once a file is compiled and an executable file is produced, you can execute, debug, or profile the program on your system. Executables produced by the PGI compilers are unconstrained, meaning they can be executed on any compatible x86 or x64 processor-based system, regardless of whether the PGI compilers are installed on that system.

Overview

In general, using a PGI compiler involves three steps:

1. Produce a program source code in a file containing a `.f` extension or another appropriate extension, as described in "Input Files," on page 4. This program may be one that you have written or one that you are modifying.
2. Compile the program using the appropriate compiler command.
3. Execute, debug, or profile the executable file on your system.

You might also want to deploy your application, though this is not a required step.

The PGI compilers allow many variations on these general program development steps. These variations include the following:

- Stop the compilation after preprocessing, compiling or assembling to save and examine intermediate results.
- Provide options to the driver that control compiler optimization or that specify various features or limitations.
- Include as input intermediate files such as preprocessor output, compiler output, or assembler output.

Invoking the Command-level PGI Compilers

To translate and link a Fortran, C, or C++ program, the pgf77, pgf95, pghpf, pgcc, and pgcpp commands do the following:

1. Preprocess the source text file.
2. Check the syntax of the source text.
3. Generate an assembly language file.
4. Pass control to the subsequent assembly and linking steps.

Example 1.1. Hello program

Let's look at a simple example of using the PGI compiler to create, compile, and execute a program that prints *hello*.

Step 1: Create your program.

For this example, suppose you enter the following simple Fortran program in the file `hello.f`:

```
print *, "hello"
end
```

Step 2: Compile the program.

When you created your program, you called it `hello.f`. In this example, we compile it from a shell command prompt using the default pgf95 driver option. Use the following syntax:

```
PGI$ pgf95 hello.f
PGI$
```

By default, the executable output is placed in the file `a.out`, or, on Windows platforms, in a filename based on the name of the first source or object file on the command line. However, you can specify an output file name by using the `-o` option.

To place the executable output in the file `hello`, use this command:

```
PGI$ pgf95 -o hello hello.f
PGI$
```

Step 3: Execute the program.

To execute the resulting hello program, simply type the filename at the command prompt and press the Return or Enter key on your keyboard:

```
PGI$ hello
 hello
PGI$
```

Command-line Syntax

The compiler command-line syntax, using pgf95 as an example, is:

```
pgf95 [options] [path]filename [...]
```

Where:

options
: is one or more command-line options, all of which are described in detail in Chapter 2, "*Using Command Line Options*".

path
: is the pathname to the directory containing the file named by filename. If you do not specify the path for a filename, the compiler uses the current directory. You must specify the path separately for each filename not in the current directory.

filename
: is the name of a source file, preprocessed source file, assembly-language file, object file, or library to be processed by the compilation system. You can specify more than one [path]filename.

Command-line Options

The command-line options control various aspects of the compilation process. For a complete alphabetical listing and a description of all the command-line options, refer to Chapter 2, "*Using Command Line Options*".

The following list provides important information about proper use of command-line options.

- Case is significant for options and their arguments.
- The compiler drivers recognize characters preceded by a hyphen (-) as command-line options. For example, the `-Mlist` option specifies that the compiler creates a listing file.

  #### Note

  The convention for the text of this manual is to show command-line options using a dash instead of a hyphen; for example, you see `-Mlist`.

- The PGC++ command recognizes a group of characters preceded by a plus sign (+) as command-line options.
- The order of options and the filename is flexible. That is, you can place options before and after the filename argument on the command line. However, the placement of some options is significant, such as the `-l` option, in which the order of the filenames determines the search order.

  #### Note

  If two or more options contradict each other, the *last* one in the command line takes precedence.

Fortran Directives and C/C++ Pragmas

You can insert Fortran directives and C/C++ pragmas in program source code to alter the effects of certain command-line options and to control various aspects of the compilation process for a specific routine or a specific program loop. For more information on Fortran directives and C/C++ pragmas, refer to Chapter 5, "*Using OpenMP*" and Chapter 8, "*Using Directives and Pragmas*".

Filename Conventions

The PGI compilers use the filenames that you specify on the command line to find and to create input and output files. This section describes the input and output filename conventions for the phases of the compilation process.

Input Files

You can specify assembly-language files, preprocessed source files, Fortran/C/C++ source files, object files, and libraries as inputs on the command line. The compiler driver determines the type of each input file by examining the filename extensions.

Note

For systems with a case-insensitive file system, use the `-M`preprocess option, described in Chapter 16, "*Command-Line Options Reference*", under the commands for Fortran preprocessing.

The drivers use the following conventions:

filename.f
: indicates a Fortran source file.

filename.F
: indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.FOR
: indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.F95
: indicates a Fortran 90/95 source file that can contain macros and preprocessor directives (to be preprocessed).

filename.f90
: indicates a Fortran 90/95 source file that is in freeform format.

filename.f95
: indicates a Fortran 90/95 source file that is in freeform format.

filename.hpf
: indicates an HPF source file.

filename.c
: indicates a C source file that can contain macros and preprocessor directives (to be preprocessed).

filename.i
: indicates a preprocessed C or C++ source file.

filename.C
: indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.cc
: indicates a C++ source file that can contain macros and preprocessor directives (to be preprocessed).

filename.s
: indicates an assembly-language file.

filename.o
: (Linux, Mac OS X, SFU, SUA) indicates an object file.

filename.obj
: (Windows systems only) indicates an object file.

filename.a
: (Linux, Mac OS X, SFU, SUA) indicates a library of object files.

filename.lib
: (Windows systems only) indicates a statically-linked library of object files or an import library.

filename.so
: (Linux and SFU systems only) indicates a library of shared object files.

filename.dll
: (Windows systems only) indicates a dynamically-linked library.

filename.dylib
: (Mac OS X systems only) indicates a dynamically-linked library.

The driver passes files with `.s` extensions to the assembler and files with .o, .obj, .so, .dll, .a and .lib extensions to the linker. Input files with unrecognized extensions, or no extension, are also passed to the linker.

Files with a `.F` (Capital F) or `.FOR` suffix are first preprocessed by the Fortran compilers and the output is passed to the compilation phase. The Fortran preprocessor functions like cpp for C/C++ programs, but is built in to the Fortran compilers rather than implemented through an invocation of cpp. This design ensures consistency in the preprocessing step regardless of the type or revision of operating system under which you're compiling.

Any input files not needed for a particular phase of processing are not processed. For example, if on the command line you specify an assembly-language file (filename.s) and the `-S` option to stop before the assembly phase, the compiler takes no action on the assembly language file. Processing stops after compilation and the assembler does not run. In this scenario, the compilation must have been completed in a previous pass which created the `.s` file. For a complete description of the `-S` option, refer to the following section:"Output Files".

In addition to specifying primary input files on the command line, code within other files can be compiled as part of include files using the `INCLUDE` statement in a Fortran source file or the `preprocessor #include` directive in Fortran source files that use a `.F` extension or C and C++ source files.

When linking a program with a library, the linker extracts only those library components that the program needs. The compiler drivers link in several libraries by default. For more information about libraries, refer to Chapter 9, "*Creating and Using Libraries*".

Output Files

By default, an executable output file produced by one of the PGI compilers is placed in the file a.out, or, on Windows, in a filename based on the name of the first source or object file on the command line. As the example in the preceding section shows, you can use the `-o` option to specify the output file name.

If you use one of the options: `-F` (Fortran only), `-P` (C/C++ only), `-S` or `-c`, the compiler produces a file containing the output of the last completed phase for each input file, as specified by the option supplied. The

output file will be a preprocessed source file, an assembly-language file, or an unlinked object file respectively. Similarly, the `-E` option does not produce a file, but displays the preprocessed source file on the standard output. Using any of these options, the `-o` option is valid only if you specify a single input file. If no errors occur during processing, you can use the files created by these options as input to a future invocation of any of the PGI compiler drivers. The following table lists the stop-after options and the output files that the compilers create when you use these options. It also describes the accepted input files.

Table 1.1. Stop-after Options, Inputs and Outputs

Option	Stop after	Input	Output
–E	preprocessing	Source files.	preprocessed file to standard out
–F	preprocessing	Source files. This option is not valid for pgcc or pgcpp.	preprocessed file (.f)
–P	preprocessing	Source files. This option is not valid for pgf77, pgf95 or pghpf)	preprocessed file (.i)
–S	compilation	Source files or preprocessed files.	assembly-language file (.s)
–c	assembly	Source files, preprocessed files or assembly-language files.	unlinked object file (.o or .obj)
none	linking	Source files, preprocessed files, assembly-language files, object files or libraries.	executable file (a.out or .exe)

If you specify multiple input files or do not specify an object filename, the compiler uses the input filenames to derive corresponding default output filenames of the following form, where filename is the input filename without its extension:

filename.f
: indicates a preprocessed file, if you compiled a Fortran file using the `-F` option.

filename.i
: indicates a preprocessed file, if you compiled using the `-P` option.

filename.lst
: indicates a listing file from the `-Mlist` option.

filename.o or filename.obj
: indicates an object file from the `-c` option.

filename.s
: indicates an assembly-language file from the `-S` option.

Note

Unless you specify otherwise, the destination directory for any output file is the current working directory. If the file exists in the destination directory, the compiler overwrites it.

The following example demonstrates the use of output filename extensions.

```
$ pgf95 -c proto.f proto1.F
```

This produces the output files proto.o and proto1.o, or, on Windows, proto.obj and proto1.obj all of which are binary object files. Prior to compilation, the file proto1.F is preprocessed because it has a .F filename extension.

Fortran, C, and C++ Data Types

The PGI Fortran, C, and C++ compilers recognize scalar and aggregate data types. A scalar data type holds a single value, such as the integer value 42 or the real value 112.6. An aggregate data type consists of one or more scalar data type objects, such as an array of integer values.

For information about the format and alignment of each data type in memory, and the range of values each type can have on x86 or x64 processor-based systems running a 32-bit operating system, refer to Chapter 15, "*Fortran, C, and C++ Data Types*".

For more information on x86-specific data representation, refer to the *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).

This manual specifically does not address x64 processor-based systems running a 64-bit operating system, because the application binary interface (ABI) for those systems is still evolving. For the latest version of this ABI, see *www.x86-64.org/abi.pdf*.

Parallel Programming Using the PGI Compilers

The PGI compilers support three styles of parallel programming:

- Automatic shared-memory parallel programs compiled using the `-Mconcur` option to pgf77, pgf95, pgcc, or pgcpp — parallel programs of this variety can be run on shared-memory parallel (SMP) systems such as dual-core or multi-processor workstations.
- OpenMP shared-memory parallel programs compiled using the `-mp` option to pgf77, pgf95, pgcc, or pgcpp — parallel programs of this variety can be run on SMP systems. Carefully coded user-directed parallel programs using OpenMP directives can often achieve significant speed-ups on dual-core workstations or large numbers of processors on SMP server systems. Chapter 5, "*Using OpenMP*" contains complete descriptions of user-directed parallel programming.
- Data parallel shared- or distributed-memory parallel programs compiled using the PGHPF High Performance Fortran compiler — parallel programs of this variety can be run on SMP workstations or servers, distributed-memory clusters of workstations, or clusters of SMP workstations or servers. Coding a data parallel version of an application can be more work than using OpenMP directives, but has the advantage that the resulting executable is usable on all types of parallel systems regardless of whether shared memory is available. See the PGHPF User's Guide for a complete description of how to build and execute data parallel HPF programs.

In this manual, the first two types of parallel programs are collectively referred to as SMP parallel programs. The third type is referred to as a data parallel program, or simply as an HPF program.

On a single silicon die, some newer CPUs incorporate two or more complete processor cores - functional units, registers, level 1 cache, level 2 cache, and so on. These CPUs are known as multi-core processors. For

purposes of HPF, threads, or OpenMP parallelism, these cores function as two or more distinct processors. However, the processing cores are on a single chip occupying a single socket on a system motherboard. For purposes of PGI software licensing, a multi-core processor is treated as a single CPU.

Running SMP Parallel Programs

When you execute an SMP parallel program, by default it uses only one processor. To run on more than one processor, set the `NCPUS` environment variable to the desired number of processors, subject to a maximum of four for PGI's workstation-class products. For information on how to set environment variables, refer to"Setting Environment Variables," on page 127

Note

If you set NCPUS to a number larger than the number of physical processors, your program may execute very slowly.

Running Data Parallel HPF Programs

When you execute an HPF program, by default it will use only one processor. If you wish to run on more than one processor, use the -pghpf -np run-time option. For example, to compile and run the hello.f example defined in "Hello program," on page 2 on one processor, you would issue the following commands:

```
% pghpf -o hello hello.f
Linking:
% hello
 hello
%
```

To execute it on two processors, you would issue the following commands:

```
% hello -pghpf -np 2
 hello
%
```

Note

If you specify a number larger than the number of physical processors, your program will execute very slowly.

You still only see a single "hello" printed to your screen. This is because HPF is a single-threaded model, meaning that all statements execute with the same semantics as if they were running in serial. However, parallel statements or constructs operating on explicitly distributed data are in fact executed in parallel. The programmer must manually insert compiler directives to cause data to be distributed to the available processors. See the PGHPF User's Guide and The High Performance Fortran Handbook for more details on constructing and executing data parallel programs on shared-memory or distributed-memory cluster systems using PGHPF.

Platform-specific considerations

There are nine platforms supported by the PGI Workstation and PGI Server compilers and tools:

- 32-bit Linux – supported on 32-bit Linux operating systems running on either a 32-bit x86 compatible or an x64 compatible processor.

- 4-bit/32-bit Linux – includes all features and capabilities of the 32-bit Linux version, and is also supported on 64-bit Linux operating systems running on an x64 compatible processor.
- 32-bit Windows – supported on 32-bit Windows operating systems running on either a 32-bit x86 compatible or an x64-compatible processor.
- 64-bit/32-bit Windows – includes all features and capabilities of the 32-bit Windows version; also supported on 64-bit Windows operating systems running an x64- compatible processor.
- 32-bit SFU – supported on 32-bit Windows operating systems running on either a 32-bit x86 compatible or an x64 compatible processor.
- 32-bit SUA – supported on 32-bit Windows operating systems running on either a 32-bit x86 compatible or an x64 compatible processor.
- 64-bit/32-bit SUA – includes all features and capabilities of the 32-bit SUA version; also supported on 64-bit Windows operating systems running on an x64-compatible processor.
- 32-bit Mac OS X – supported on 32-bit Mac OS X operating systems running on either a 32-bit or 64-bit Intel-based Mac system.
- 64-bit Mac OS X – supported on 64-bit Mac OS X operating systems running on a 64-bit Intel-based Mac system.

The following sections describe the specific considerations required to use the PGI compilers on the various platforms: Linux, Windows, and Mac OS X.

Using the PGI Compilers on Linux

Linux Header Files

The Linux system header files contain many GNU gcc extensions. PGI supports many of these extensions, thus allowing the PGCC C and C++ compilers to compile most programs that the GNU compilers can compile. A few header files not interoperable with the PGI compilers have been rewritten. These files are included in `$PGI/linux86/include`, such as `sigset.h`, `asm/byteorder.h`, `stddef.h`, `asm/posix_types.h` and others. Also, PGI's version of `stdarg.h` supports changes in newer versions of Linux.

If you are using the PGCC C or C++ compilers, please make sure that the supplied versions of these include files are found before the system versions. This will happen by default unless you explicitly add a –I option that references one of the system include directories.

Running Parallel Programs on Linux

You may encounter difficulties running auto-parallel or OpenMP programs on Linux systems when the per-thread stack size is set to the default (2MB). If you have unexplained failures, please try setting the environment variable `OMP_STACKSIZE` to a larger value, such as 8MB. For information on setting environment variables, refer to "Setting Environment Variables," on page 127.

If your program is still failing, you may be encountering the hard 8 MB limit on main process stack sizes in Linux. You can work around the problem by issuing the following command in csh:

```
% limit stacksize unlimited
```

In bash, sh, zsh, or ksh, use:

```
$ ulimit -s unlimited
```

Using the PGI Compilers on Windows

A UNIX-like shell environment, Cygwin, is bundled with PGI Workstation for Windows to provide a familiar development environment for Linux or UNIX users. PGI Workstation for SFU and SUA does not include Cygwin; it runs in the SFU/SUA shell environment.

BASH Shell Environment

On Windows platforms, the tools that ship with the PGI Workstation or PGI Server command-level compilers include a full-featured shell command environment. After installation, you have a PGI Workstation icon on your Windows desktop. Double-left-click on this icon to launch an instance of the Cygwin BASH command shell window. Working within BASH is very much like working within the sh or ksh shells on a Linux system, but in addition BASH has a command history feature similar to csh and several other unique features. Shell programming is fully supported. A complete BASH User's Guide is available through the PGI online manual set. Select "PGI Workstation" under Start->Programs and double-left-click on the documentation icon to see the online manual set. To read the online manuals, you must have a web browser installed on your system.

The BASH shell window is pre-initialized for usage of the PGI compilers and tools, so there is no need to set environment variables or modify your command path when the command window comes up. In addition to the PGI compiler commands, within BASH you have access to over 100 common commands and utilities, including but not limited to the following:

vi	gzip / gunzip	ftp
tar / untar	grep / egrep / fgrep	awk
sed	cksum	cp
cat	diff	du
date	kill	ls
find	mv	printenv / env
more / less	touch	wc
rm / rmdir	make	

If you are familiar with program development in a Linux environment, editing, compiling, and executing programs within BASH will be very comfortable. If you have not previously used such an environment, you should take time to familiarize yourself with *vi* or other editors and with makefiles. The World Wide Web has an extensive online tutorial available for the *vi* editor as well as a number of thorough introductions to the construction and use of makefiles.

On Win64, the desktop icon starts a bash shell configured for building 64-bit programs. To start a bash shell configured for building 32-bit programs, launch PGI Workstation (32-bit) from the Start menu.

Alternatively, by selecting the appropriate option from the PGI Workstation program group accessed in the usual way through the "Start" menu, you can launch a standard Windows command window that is pre-initialized to enable use of the PGI compilers and tools.

Note

The command window launched by PGI Workstation can be customized using the "Properties" selection on the menu accessible by right-clicking the window's title bar.

For library compatibility, PGI provides versions of **ar** and **ranlib** that are compatible with native Windows object-file formats. For more information on these commands, refer to "Creating and Using Static Libraries on Windows," on page 116.

Windows Command Prompt

The PGI Workstation entry in the Windows Start menu contains a submenu titled PGI Workstation Tools. This submenu contains a shortcut labeled PGI Command Prompt (32-bit). You use this shortcut to launch a Windows command shell using an environment pre-initialized for the use of the 32-bit PGI compilers and tools. On x64 systems, a second shortcut labeled PGI Command Prompt (64-bit) is also present. This shortcut launches a Windows command shell using an environment pre-initialized for the use of the 64-bit PGI compilers and tools.

Using the PGI Compilers on SUA and SFU

Subsystem for Unix Applications (SUA and SFU)

Subsystem for Unix Applications (SUA) is a source-compatibility subsystem for running Unix applications on 32-bit and 64-bit Windows server-class operating systems. PGI Workstation for Windows includes compilers and tools for SUA and its 32-bit-only predecessor, Services For Unix (SFU).

SUA provides an operating system for POSIX processes. There is a package of support utilities available for download from Microsoft that provides a more complete Unix environment, including features like shells, scripting utilities, a telnet client, development tools, and so on.

SUA/SFU Header Files

The SUA/SFU system header files contain numerous non-standard extensions. PGI supports many of these extensions, thus allowing the PGCC C and C++ compilers to compile most programs that the GNU compilers can compile. A few header files not interoperable with the PGI compilers have been rewritten and are included in $PGI/sua32/include or $PGI/sua64/include. These files are: stdarg.h, stddef.h, and others.

If you are using the PGCC C or C++ compilers, please make sure that the supplied versions of these include files are found before the system versions. This happens by default unless you explicitly add a –I option that references one of the system include directories.

Running Parallel Programs on SUA and SFU

You may encounter difficulties running auto-parallel or OpenMP programs on SUA/SFU systems when the per-thread stack size is set to the default (2MB). If you have unexplained failures, please try setting the environment variable `OMP_STACKSIZE` to a larger value, such as 8MB. For information on how to set environment variables, refer to "Setting Environment Variables," on page 127.

Using Shared object files in SFU and SUA

PGI Workstation for 32-bit SFU and 32-bit SUA uses the GNU ld for its linker. Further, the PGI compilers and tools are able to generate shared object (.so) files. You use the -shared switch to generate a shared object file.

For examples of how to create a shared object file and a program that uses it, as well as compiler switches that support shared objects, refer to "Creating and Using Shared Object Files in SFU and 32-bit SUA," on page 114.

Using the PGI Compilers on Mac OS X

PGI Workstation 9.0 for Mac OS X supports most of the features of the 32-and 64-bit versions for linux86 and linux86-64 environments. Typically the PGI compilers and tools on Mac OS X function identically to their Linux counterparts.

Mac OS X Header Files

The Mac OS X header files contain numerous non-standard extensions. PGI supports many of these extensions, thus allowing the PGCC C and C++ compilers to compile most programs that the GNU compilers can compile. A few header files not interoperable with the PGI compilers have been rewritten. These files are included in $PGI/sua32/include or $PGI/sua64/include. These files are: stdarg.h, stddef.h, and others.

If you are using the PGCC C or C++ compilers, please make sure that the supplied versions of these include files are found before the system versions. This will happen by default unless you explicitly add a –I option that references one of the system include directories.

Mac OS Debugging Requirements

Both the `-g` and `-Mkeepobj` switches play important roles when compiling a program on Apple Mac OS for debugging.

- To debug a program with symbol information on the Mac OS, files must be compiled with the -g switch to keep the program's object files, the files with a ".o" extension. Further, these object files must remain in the same directory in which they were created.
- If a program is built with separate compile and link steps, by compiling with the -c switch which generates the ".o" object files, then using the –g switch guarantees the required object files are available for debugging.

Use the following command sequence to compile and then link your code.

To compile the programs, use these commands:

```
pgcc -c -g main.cpgcc -c -g foo.cpgcc -c -g bar.c
```

To link, use this command:

```
pgcc -g main.o foo.o bar.o
```

Linking on Mac OS X

On the Mac OS X, the PGI Workstation 9.0 compilers do not support static linking of user binaries. For compatibility with future Apple updates, the compilers support dynamic linking of user binaries. For more information on dynamic linking, refer to "Creating and Using Dynamic Libraries on Mac OS X," on page 115.

Running Parallel Programs on Mac OS X

You may encounter difficulties running auto-parallel or OpenMP programs on Mac OS X systems when the per-thread stack size is set to the default (8MB). If you have unexplained failures, please try setting the environment variable OMP_STACKSIZE to a larger value, such as 16MB. For information on how to set environment variables, refer to "Setting Environment Variables," on page 127.

Site-specific Customization of the Compilers

If you are using the PGI compilers and want all your users to have access to specific libraries or other files, there are special files that allow you to customize the compilers for your site.

Using siterc Files

The PGI compiler drivers utilize a file named `siterc` to enable site-specific customization of the behavior of the PGI compilers. The `siterc` file is located in the `bin` subdirectory of the PGI installation directory. Using `siterc`, you can control how the compiler drivers invoke the various components in the compilation tool chain.

Using User rc Files

In addition to the siterc file, user `rc` files can reside in a given user's home directory, as specified by the user's HOME environment variable. You can use these files to control the respective PGI compilers. All of these files are optional.

On Linux, SUA, and Mac OS X, these files are named `.mypgf77rc`, `.mypgf90rc`, `.mypgccrc`, `.mypgcpprc`, and `.mypghpfrc`.

On Windows, these files are named `mypgf77rc, mmupgf90rc, ypgf95rc, mypgccrc, mypgcpprc`, and `mypghpfrc`.

The following examples show how these `rc` files can be used to tailor a given installation for a particular purpose.

Table 1.2. Examples of Using siterc and User rc Files

To do this...	Add the line shown to the indicated file
Make available to all linux86-64 compilations the libraries found in `/opt/newlibs/64`	`set SITELIB=/opt/newlibs/64;` to `/opt/pgi/linux86-64/9.0/bin/siterc`
Make available to all linux86 compilations the libraries found in `/opt/newlibs/32`	`set SITELIB=/opt/newlibs/32;` to `/opt/pgi/linux86/9.0/bin/siterc`
Add to all linux86-64 compilations a new library path: `/opt/local/fast`	`append SITELIB=/opt/local/fast;` to `/opt/pgi/linux86-64/9.0/bin/siterc`

To do this...	Add the line shown to the indicated file
Make available to all compilations the include path `-I/opt/acml/include`	`set SITEINC=/opt/acml/include;` to `/opt/pgi/linux86/9.0/bin/siterc` and `/opt/pgi/linux86-64/9.0/bin/siterc`

With linux86-64 compilations, change –Mmpi to link in
 	
`/opt/mympi/64/libmpix.a`
	`set MPILIBDIR=/opt/mympi/64;`
	`set MPILIBNAME=mpix;`
	
	to `/opt/pgi/linux86-64/9.0/bin/siterc;`

Have linux86-64 compilations always add `–DIS64BIT –DAMD`	`set SITEDEF=IS64BIT AMD;` to `/opt/pgi/linux86-64/9.0/bin/siterc`
Build an F90 executable for linux86-64 or linux86 that resolves PGI shared objects in the relative directory `./REDIST`	`set RPATH=./REDIST ;` to `~/.mypgf95rc` **Note.** This only affects the behavior of PGF95 for the given user.

Common Development Tasks

Now that you have a brief introduction to the compiler, let's look at some common development tasks that you might wish to perform.

- When you compile code you can specify a number of options on the command line that define specific characteristics related to how the program is compiled and linked, typically enhancing or overriding the default behavior of the compiler. For a list of the most common command line options and information on all the command line options, refer to Chapter 2, "*Using Command Line Options*".
- Code optimization and parallelization allows the compiler to organize your code for efficient execution. While possibly increasing compilation time and making the code more difficult to debug, these techniques typically produce code that runs significantly faster than code that does not use them. For more information on optimization and parallelization, refer to Chapter 3, "*Optimizing & Parallelizing*".
- Function inlining, a special type of optimization, replaces a call to a function or a subroutine with the body of the function or subroutine. This process can speed up execution by eliminating parameter passing and the function or subroutine call and return overhead. In addition, function inlining allows the compiler to optimize the function with the rest of the code. However, function inlining may also result in much larger code size with no increase in execution speed. For more information on function inlining, refer to Chapter 4, "*Using Function Inlining*".
- Directives and pragmas allow users to place hints in the source code to help the compiler generate better assembly code. You typically use directives and pragmas to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. You place them in your source code where you want them to take effect. A directive or pragma typically stays in effect from the point where included until the end of the compilation unit or until another directive or pragma changes its status. For more information on directives and pragmas, refer to Chapter 5, "*Using OpenMP*" and Chapter 8, "*Using Directives and Pragmas*".

- A library is a collection of functions or subprograms used to develop software. Libraries contain "helper" code and data, which provide services to independent programs, allowing code and data to be shared and changed in a modular fashion. The functions and programs in a library are grouped for ease of use and linking. When creating your programs, it is often useful to incorporate standard libraries or proprietary ones. For more information on this topic, refer to Chapter 9, "*Creating and Using Libraries*".
- Environment variables define a set of dynamic values that can affect the way running processes behave on a computer. It is often useful to use these variables to set and pass information that alters the default behavior of the PGI compilers and the executables which they generate. For more information on these variables, refer to Chapter 10, " *Using Environment Variables*".
- Deployment, though possibly an infrequent task, can present some unique issues related to concerns of porting the code to other systems. Deployment, in this context, involves distribution of a specific file or set of files that are already compiled and configured. The distribution must occur in such a way that the application executes accurately on another system which may not be configured exactly the same as the system on which the code was created. For more information on what you might need to know to successfully deploy your code, refer to Chapter 11, "*Distributing Files - Deployment*".
- An intrinsic is a function available in a given language whose implementation is handled specially by the compiler. Intrinsics make using processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages details that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data. For C/C++ programs, PGI provides support for MMX and SSE/SSE2/SSE3 intrinsics. For more information on these intrinsics, refer to Chapter 24, "*C/C++ MMX/SSE Inline Intrinsics*".

Chapter 2. Using Command Line Options

A command line option allows you to control specific behavior when a program is compiled and linked. This chapter describes the syntax for properly using command-line options and provides a brief overview of a few of the more common options.

Note

For a complete list of command-line options, their descriptions and use, refer to Chapter 16, "*Command-Line Options Reference*," on page 195.

Command Line Option Overview

Before looking at all the command-line options, first become familiar with the syntax for these options. There are a large number of options available to you, yet most users only use a few of them. So, start simple and progress into using the more advanced options.

By default, the PGI compilers generate code that is optimized for the type of processor on which compilation is performed, the compilation host. Before adding options to your command-line, review the sections "Help with Command-line Options," on page 18 and "Frequently-used Options," on page 21.

Command-line Options Syntax

On a command-line, options need to be preceded by a hyphen (-). If the compiler does not recognize an option, it passes the option to the linker.

This document uses the following notation when describing options:

[item]
: Square brackets indicate that the enclosed item is optional.

{item | item}
: Braces indicate that you must select one and only one of the enclosed items. A vertical bar (|) separates the choices.

...

Horizontal ellipses indicate that zero or more instances of the preceding item are valid.

NOTE

Some options do not allow a space between the option and its argument or within an argument. When applicable, the syntax section of the option description in Chapter 16, "*Command-Line Options Reference*," on page 195 contains this information.

Command-line Suboptions

Some options accept several suboptions. You can specify these suboptions either by using the full option statement multiple times or by using a comma-separated list for the suboptions.

The following two command lines are equivalent:

```
pgf95 -Mvect=sse -Mvect=noaltcode
```

```
pgf95 -Mvect=sse,noaltcode
```

Command-line Conflicting Options

Some options have an opposite or negated counterpart. For example, both `-Mvect` and `-Mnovect` are available. `-Mvect` enables vectorization and `-Mnovect` disables it. If you used both of these commands on a command line, they would conflict.

Note

Rule: When you use conflicting options on a command line, the last encountered option takes precedence over any previous one.

This rule is important for a number of reasons.

- Some options, such as `-fast`, include other options. Therefore, it is possible for you to be unaware that you have conflicting options.
- You can use this rule to create makefiles that apply specific flags to a set of files, as shown in Example 2.1.

Example 2.1. Makefiles with Options

In this makefile fragment, CCFLAGS uses vectorization. CCNOVECTFLAGS uses the flags defined for CCFLAGS but disables vectorization.

```
CCFLAGS=c -Mvect=sse
CCNOVECTFLAGS=$(CCFLAGS) -Mnovect
```

Help with Command-line Options

If you are just getting started with the PGI compilers and tools, it is helpful to know which options are available, when to use them, and which options most users find effective.

Using –help

The `-help` option is useful because it provides information about all options supported by a given compiler. You can use `-help` in one of three ways:

- Use `–help` with no parameters to obtain a list of all the available options with a brief one-line description of each.
- Add a parameter to `–help` to restrict the output to information about a specific option. The syntax for this usage is this:

```
–help <command line option>
```

Suppose you use the following command to restrict the output to information about the `-fast` option:

```
$ pgf95 -help -fast
```

The output you see is similar to this:

```
-fast Common optimizations; includes -O2 -Munroll=c:1 -Mnoframe -Mlre
```

In the following example, we add the `–help` parameter to restrict the output to information about the help command. The usage information for `–help` shows how groups of options can be listed or examined according to function.

```
$ pgf95 -help -help
-help[=groups|asm|debug|language|linker|opt|other|
 overall|phase|prepro|suffix|switch|target|variable]
 Show compiler switches
```

- Add a parameter to `–help` to restrict the output to a specific set of options or to a building process. The syntax for this usage is this:

```
-help=<subgroup>
```

By using the command `pgf95 -help -help`, as previously shown, we can see output that shows the available subgroups. You can use the following command to restrict the output on the `–help` command to information about only the options related to only one group, such as debug information generation.

```
$ pgf95 -help=debug
```

The output you see is similar to this:

```
Debugging switches:
-M[no]bounds Generate code to check array bounds
-Mchkfpstk Check consistency of floating point stack at subprogram calls
 (32-bit only)
-Mchkstk Check for sufficient stack space upon subprogram entry
-Mcoff Generate COFF format object
-Mdwarf1 Generate DWARF1 debug information with -g
-Mdwarf2 Generate DWARF2 debug information with -g
-Mdwarf3 Generate DWARF3 debug information with -g
-Melf Generate ELF format object
-g Generate information for debugger
-gopt Generate information for debugger without disabling
 optimizations
```

For a complete description of subgroups, refer to “–help ,” on page 212.

Getting Started with Performance

One of the top priorities of most users is performance and optimization. This section provides a quick overview of a few of the command-line options that are useful in improving performance.

Using –fast and –fastsse Options

PGI compilers implement a wide range of options that allow users a fine degree of control on each optimization phase. When it comes to optimization of code, the quickest way to start is to use the options `-fast` and `-fastsse`. These options create a generally optimal set of flags for targets that support SSE/SSE2 capability. They incorporate optimization options to enable use of vector streaming SIMD (SSE/SSE2) instructions for 64-bit targets. They enable vectorization with SSE instructions, cache alignment, and SSE arithmetic to flush to zero mode.

Note

The contents of the `-fast` and `-fastsse` options are host-dependent. Further, you should use these options on both compile and link command lines.

- `-fast` and `-fastsse` typically include these options:

–O2	Specifies a code optimization level of 2.
–Munroll=c:1	Unrolls loops, executing multiple instances of the loop during each iteration.
–Mnoframe	Indicates to not generate code to set up a stack frame.
–Mlre	Indicates loop-carried redundancy elimination.

- These additional options are also typically available when using `-fast` for 64-bit targets or `-fastsse` for both 32- and 64-bit targets:

–Mvect=sse	Generates SSE instructions.
–Mscalarsse	Generates scalar SSE code with xmm registers; implies –Mflushz.
–Mcache_align	Aligns long objects on cache-line boundaries.
–Mflushz	Sets SSE to flush-to-zero mode.
–M[no]vect	Controls automatic vector pipelining.

Note

For best performance on processors that support SSE instructions, use the PGF95 compiler, even for FORTRAN 77 code, and the `-fast` option.

To see the specific behavior of `-fast` for your target, use the following command:

```
$ pgf95 -help -fast
```

Other Performance-related Options

While `-fast` and `-fastsse` are options designed to be the quickest route to best performance, they are limited to routine boundaries. Depending on the nature and writing style of the source code, the compiler often can perform further optimization by knowing the global context of usage of a given routine. For instance, determining the possible value range of actual parameters of a routine could enable a loop to be vectorized; similarly, determining static occurrence of calls helps to decide which routine is beneficial to inline.

These types of global optimizations are under control of Interprocedural Analysis (IPA) in PGI compilers. Option `-Mipa` enables Interprocedural Analysis. `-Mpi=fast` is the recommended option to get best performances for global optimization. You can also add the suboption `inline` to enable automatic global inlining across files. You might consider using `-Mipa=fast,inline`. This option for interprocedural analysis and global optimization can improve performance.

You may also obtain further performance improvements by experimenting with the `-M<pgflag>` options described in the section "–M Options by Category," on page 253. These options include, but are not limited to, `-Mconcur`, `-Mvect`, `-Munroll`, `-Minline`, and `-Mpfi/-Mpfo`. However, performance improvements using these options are typically application- and system-dependent. It is important to time your application carefully when using these options to ensure no performance degradations occur.

For more information on optimization, refer to Chapter 3, "*Optimizing & Parallelizing*," on page 23. For specific information about these options, refer to "Optimization Controls," on page 264.

Targeting Multiple Systems - Using the -tp Option

The `-tp` option allows you to set the target architecture. By default, the PGI compiler uses all supported instructions wherever possible when compiling on a given system. As a result, executables created on a given system may not be usable on previous generation systems. For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.

Processor-specific optimizations can be specified or limited explicitly by using the `-tp` option. Thus, it is possible to create executables that are usable on previous generation systems. With the exception of k8-64, k8-64e, p7-64, and x64, any of these sub-options are valid on any x86 or x64 processor-based system. The k8-64, k8-64e, p7-64 and x64 options are valid only on x64 processor-based systems

For more information about the `-tp` option, refer to "–tp <target> [,target...]," on page 236.

Frequently-used Options

In addition to overall performance, there are a number of other options that many users find useful when getting started. The following table provides a brief summary of these options.

For more information on these options, refer to the complete description of each option available in Chapter 16, "*Command-Line Options Reference*," on page 195. Also, there are a number of suboptions available with each of the `-M` options listed. For more information on those options, refer to "–M Options by Category".

Table 2.1. Commonly Used Command Line Options

Option	Description
–fast –fastsse	These options create a generally optimal set of flags for targets that support SSE/SSE2 capability. They incorporate optimization options to enable use of vector streaming SIMD instructions (64-bit targets) and enable vectorization with SEE instructions, cache aligned and flushz.

Option	Description
–g	Instructs the compiler to include symbolic debugging information in the object module.
–gopt	Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when –g is not specified.
–help	Provides information about available options.
–mcmodel=medium	Enables medium=model core generation for 64-bit targets; useful when the data space of the program exceeds 4GB.
–Mconcur	Instructs the compiler to enable auto-concurrentization of loops. If specified, the compiler uses multiple processors to execute loops that it determines to be parallelizable; thus, loop iterations are split to execute optimally in a multithreaded execution context.
–Minfo	Instructs the compiler to produce information on standard error.
–Minline	Enables function inlining.
–Mipa=fast,inline	Enables interprocedural analysis and optimization. Also enables automatic procedure inlining.
–Mpfi or –Mpfo	Enable profile feedback driven optimizations.
–Mkeepasm	Keeps the generated assembly files.
–Munroll	Invokes the loop unroller to unroll loops, executing multiple instances of the loop during each iteration. This also sets the optimization level to 2 if the level is set to less than 2, or if no –O or –g options are supplied.
–M[no]vect	Enables/Disables the code vectorizer.
--[no_]exceptions	Removes exception handling from user code. For C++, declares that the functions in this file generate no C++ exceptions, allowing more optimal code generation.
–o	Names the output file.
–O<level>	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
–tp <target> [,target...]	Specify the target processor(s); for the 64-bit compilers, more than one target is allowed, and enables generation of PGI Unified Binary executables.

Chapter 3. Optimizing & Parallelizing

Source code that is readable, maintainable, and produces correct results is not always organized for efficient execution. Normally, the first step in the program development process involves producing code that executes and produces the correct results. This first step usually involves compiling without much worry about optimization. After code is compiled and debugged, code optimization and parallelization become an issue.

Invoking one of the PGI compiler commands with certain options instructs the compiler to generate optimized code. Optimization is not always performed since it increases compilation time and may make debugging difficult. However, optimization produces more efficient code that usually runs significantly faster than code that is not optimized.

Note

PGI provides a profiler, PGPROF, that provides a way to visualize the performance of the components of your program. Using tables and graphs, PGPROF associates execution time and resource utilization data with the source code and instructions of your program, allowing you to see where execution time is spent. Through resource utilization data and compiler analysis information, PGPROF helps you to understand why certain parts of your program have high execution times.

The compilers optimize code according to the specified optimization level. You can use a number of options to specify the optimization levels, including `-O`, `-Mvect`, `-Mipa`, and `-Mconcur`. In addition, you can use several of the –M<pgflag> switches to control specific types of optimization and parallelization.

This chapter describes the optimization options displayed in the following list.

`-fast`	`-Minline`	`-Mpfi`	`-Mvect`
`-Mconcur`	`-Mipa=fast`	`-Mpfo`	`-O`
`-Minfo`	`-Mneginfo`	`-Munroll`	`-Msafeptr`

This chapter also describes how to choose optimization options to use with the PGI compilers. This overview will help if you are just getting started with one of the PGI compilers, or wish to experiment with individual optimizations. Complete specifications of each of these options is available in Chapter 16, "*Command-Line Options Reference*".

Overview of Optimization

In general, optimization involves using transformations and replacements that generate more efficient code. This is done by the compiler and involves replacements that are independent of the particular target processor's architecture as well as replacements that take advantage of the x86 or x64 architecture, instruction set and registers. For the discussion in this and the following chapters, optimization is divided into the following categories:

Local Optimization

This optimization is performed on a block-by-block basis within a program's basic blocks. A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end. The PGI compilers perform many types of local optimization including: algebraic identity removal, constant folding, common sub-expression elimination, redundant load and store elimination, scheduling, strength reduction, and peephole optimizations.

Global Optimization

This optimization is performed on a program unit over all its basic blocks. The optimizer performs control-flow and data-flow analysis for an entire program unit. All loops, including those formed by IFs and GOTOs, are detected and optimized. Global optimization includes: constant propagation, copy propagation, dead store elimination, global register allocation, invariant code motion, and induction variable elimination.

Loop Optimization: Unrolling, Vectorization, and Parallelization

The performance of certain classes of loops may be improved through vectorization or unrolling options. Vectorization transforms loops to improve memory access performance and make use of packed SSE instructions which perform the same operation on multiple data items concurrently. Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions. Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers.

Interprocedural Analysis (IPA) and Optimization

Interprocedural analysis (IPA) allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable. For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the callee and perform optimizations that are not valid if the dummy argument is treated as a variable. A wide range of optimizations are enabled or improved by using IPA, including but not limited to data alignment optimizations, argument removal, constant propagation, pointer disambiguation, pure function detection, F90/F95 array shape propagation, data placement, vestigial function removal, automatic function inlining, inlining of functions from pre-compiled libraries, and interprocedural optimization of functions from pre-compiled libraries.

Function Inlining

This optimization allows a call to a function to be replaced by a copy of the body of that function. This optimization will sometimes speed up execution by eliminating the function call and return overhead. Function inlining may also create opportunities for other types of optimization. Function inlining is not always beneficial. When used improperly it may increase code size and generate less efficient code.

Profile-Feedback Optimization (PFO)

Profile-feedback optimization (PFO) makes use of information from a trace file produced by specially instrumented executables which capture and save information on branch frequency, function and subroutine call frequency, semi-invariant values, loop index ranges, and other input data dependent information that can only be collected dynamically during execution of a program. By definition, use of profile-feedback optimization is a two-phase process: compilation and execution of a specially-instrumented executable, followed by a subsequent compilation which reads a trace file generated during the first phase and uses the information in that trace file to guide compiler optimizations.

Getting Started with Optimizations

Your first concern should be getting your program to execute and produce correct results. To get your program running, start by compiling and linking without optimization. Use the optimization level `–O0` or select `–g` to perform minimal optimization. At this level, you will be able to debug your program easily and isolate any coding errors exposed during porting to x86 or x64 platforms.

If you want to get started quickly with optimization, a good set of options to use with any of the PGI compilers is `–fast –Mipa=fast`. For example:

```
$ pgf95 -fast -Mipa=fast prog.f
```

For all of the PGI Fortran, C, and C++ compilers, the `–fast –Mipa=fast` options generally produce code that is well-optimized without the possibility of significant slowdowns due to pathological cases.

- The `–fast` option is an aggregate option that includes a number of individual PGI compiler options; which PGI compiler options are included depends on the target for which compilation is performed.
- The `–Mipa=fast` option invokes interprocedural analysis including several IPA suboptions.
- For C++ programs, add `-Minline=levels:10 --no_exceptions` as shown here:

```
$ pgcpp -fast -Mipa=fast -Minline=levels:10 --no_exceptions prog.cc
```

Note

A C++ program compiled with `--no_exceptions` fails if the program uses exception handling.

By experimenting with individual compiler options on a file-by-file basis, further significant performance gains can sometimes be realized. However, depending on the coding style, individual optimizations can sometimes cause slowdowns, and must be used carefully to ensure performance improvements. In addition to `-fast`, the optimization flags most likely to further improve performance are `-O3`, `-Mpfi`, `-Mpfo`, `-Minline`; and on targets with multiple processors, you can use `-Mconcur`.

In addition, the `–Msafeptr` option can significantly improve performance of C/C++ programs in which there is known to be no pointer aliasing. For obvious reasons this command-line option must be used carefully.

Three other extremely useful options are `-help`, `-Minfo`, and `-dryrun`.

–help

As described in "Help with Command-line Options," on page 18, you can see a specification of any command-line option by invoking any of the PGI compilers with `-help` in combination with the option in question, without specifying any input files.

For example, you might want information on `-O`:

```
$ pgf95 -help -O
```

The resulting output is similar to this:

```
Reading rcfile /usr/pgi/linux86-64/7.0/bin/.pgf95rc
-O[<n>] Set optimization level, -O0 to -O4, default -O2
```

Or you can see the full functionality of `-help` itself, which can return information on either an individual option or groups of options:

```
$ pgf95 -help -help
```

The resulting output is similar to this:

```
Reading rcfile /usr/pgi_rel/linux86-64/7.0/bin/.pgf95rc
-help[=groups|asm|debug|language|linker|opt|other|overall|
  phase|prepro|suffix|switch|target|variable]
```

–Minfo

You can use the `-Minfo` option to display compile-time optimization listings. When this option is used, the PGI compilers issue informational messages to stderr as compilation proceeds. From these messages, you can determine which loops are optimized using unrolling, SSE instructions, vectorization, parallelization, interprocedural optimizations and various miscellaneous optimizations. You can also see where and whether functions are inlined.

For more information on `–Minfo`, refer to "Optimization Controls," on page 264.

–Mneginfo

You can use the `-Mneginfo` option to display informational messages listing why certain optimizations are inhibited.

For more information on `–Mneginfo`, refer to "Optimization Controls," on page 264.

–dryrun

The `–dryrun` option can be useful as a diagnostic tool if you need to see the steps used by the compiler driver to preprocess, compile, assemble and link in the presence of a given set of command line inputs. When you specify the `–dryrun` option, these steps will be printed to stderr but are not actually performed. For example, you can use this option to inspect the default and user-specified libraries that are searched during the link phase, and the order in which they are searched by the linker.

The remainder of this chapter describes the `–O` options, the loop unroller option `–Munroll`, the vectorizer option `–Mvect`, the auto-parallelization option `–Mconcur`, the interprocedural analysis optimization `–Mipa`, and the profile-feedback instrumentation (`–Mpfi`) and optimization (`–Mpfo`) options. You should be able to get very near optimal compiled performance using some combination of these switches.

Common Compiler Feedback Format (CCFF)

Using the Common Compiler Feedback Format (CCFF), PGI compilers save information about how your program was optimized, or why a particular optimization was not made, in the executable file. To append this information to the object file, use the compiler option `-Minfo=ccff`.

If you choose to use PGPROF to aid with your optimization, PGPROF can extract this information and associate it with source code and other performance data, allowing you to view all of this information simultaneously in one of the available profiler panels.

Local and Global Optimization using -O

Using the PGI compiler commands with the –Olevel option (the capital O is for Optimize), you can specify any of the following optimization levels:

–O0
: Level zero specifies no optimization. A basic block is generated for each language statement.

–O1
: Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

–O2
: Level two specifies global optimization. This level performs all level-one local optimization as well as level-two global optimization. If optimization is specified on the command line without a level, level 2 is the default.

–O3
: Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

–O4
: Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

> **Note**
>
> If you use the `-O` option to specify optimization and do not specify a level, then level-two optimization (`-O2`) is the default.

Level-zero optimization specifies no optimization (`-O0`). At this level, the compiler generates a basic block for each statement. Performance will almost always be slowest using this optimization level. This level is useful for the initial execution of a program. It is also useful for debugging, since there is a direct correlation between the program text and the code generated.

Level-one optimization specifies local optimization (`-O1`). The compiler performs scheduling of basic blocks as well as register allocation. Local optimization is a good choice when the code is very irregular, such as code that contains many short statements containing IF statements and does not contain loops (DO or DO WHILE

statements). Although this case rarely occurs, for certain types of code, this optimization level may perform better than level-two (`-O2`).

The PGI compilers perform many different types of local optimizations, including but not limited to:

- Algebraic identity removal
- Constant folding
- Common subexpression elimination
- Local register optimization
- Peephole optimizations
- Redundant load and store elimination
- Strength reductions

Level-two optimization (`-O2` or `-O`) specifies global optimization. The `-fast` option generally will specify global optimization; however, the `-fast` switch varies from release to release, depending on a reasonable selection of switches for any one particular release. The –O or –O2 level performs all level-one local optimizations as well as global optimizations. Control flow analysis is applied and global registers are allocated for all functions and subroutines. Loop regions are given special consideration. This optimization level is a good choice when the program contains loops, the loops are short, and the structure of the code is regular.

The PGI compilers perform many different types of global optimizations, including but not limited to:

- Branch to branch elimination
- Constant propagation
- Copy propagation
- Dead store elimination
- Global register allocation
- Invariant code motion
- Induction variable elimination

You can explicitly select the optimization level on the command line. For example, the following command line specifies level-two optimization which results in global optimization:

```
$ pgf95 -O2 prog.f
```

Specifying `-O` on the command-line without a level designation is equivalent to `-O2`. The default optimization level changes depending on which options you select on the command line. For example, when you select the `-g` debugging option, the default optimization level is set to level-zero (`-O0`). However, if you need to debug optimized code, you can use the `-gopt` option to generate debug information without perturbing optimization. Refer to "Default Optimization Levels," on page 44 for a description of the default levels.

As noted previously, the `-fast` option includes `-O2` on all x86 and x64 targets. If you want to override the default for`-fast` with `-O3` while maintaining all other elements of `-fast`, simply compile as follows:

```
$ pgf95 -fast -O3 prog.f
```

Scalar SSE Code Generation

For all processors prior to Intel Pentium 4 and AMD Opteron/Athlon64, for example Intel Pentium III and AMD AthlonXP/MP processors, scalar floating-point arithmetic as generated by the PGI Workstation compilers is performed using x87 floating-point stack instructions. With the advent of SSE/SSE2 instructions on Intel Pentium 4/Xeon and AMD Opteron/Athlon64, it is possible to perform all scalar floating-point arithmetic using SSE/SSE2 instructions. In most cases, this is beneficial from a performance standpoint.

The default on 32-bit Intel Pentium II/III (options `-tp p6`, `-tp piii`, and so on) or on AMD AthlonXP/MP (`option -tp k7`) is to use x87 instructions for scalar floating-point arithmetic. The default on Intel

Pentium 4/Xeon or Intel EM64T running a 32-bit operating system (`-tp p7`), AMD Opteron/Athlon64 running a 32-bit operating system (`-tp k8-32`), or AMD Opteron/Athlon64 or Intel EM64T processors running a 64-bit operating system (using `-tp k8-64` and `-tp p7-64` respectively) is to use SSE/SSE2 instructions for scalar floating-point arithmetic. The only way to override this default on AMD Opteron/Athlon64 or Intel EM64T processors running a 64-bit operating system is to specify an older 32-bit target. For example, you can use `-tp k7` or `-tp piii`.

Note

There can be significant arithmetic differences between calculations performed using x87 instructions versus SSE/SSE2.

By default, all floating-point data is promoted to IEEE 80-bit format when stored on the x87 floating-point stack, and all x87 operations are performed register-to-register in this same format. Values are converted back to IEEE 32-bit or IEEE 64-bit when stored back to memory (for REAL/float and DOUBLE PRECISION/double data respectively). The default precision of the x87 floating-point stack can be reduced to IEEE 32-bit or IEEE 64-bit globally by compiling the main program with the `-pc {32|64}` option to the PGI compilers, which is described in detail in Chapter 2, "*Using Command Line Options*". However, there is no way to ensure that operations performed in mixed precision will match those produced on a traditional load-store RISC/UNIX system which implements IEEE 64-bit and IEEE 32-bit registers and associated floating-point arithmetic instructions.

In contrast, arithmetic results produced on Intel Pentium 4/Xeon, AMD Opteron/Athlon64 or Intel EM64T processors will usually closely match or be identical to those produced on a traditional RISC/UNIX system if all scalar arithmetic is performed using SSE/SSE2 instructions. You should keep this in mind when porting applications to and from systems which support both x87 and full SSE/SSE2 floating-point arithmetic. Many subtle issues can arise which affect your numerical results, sometimes to several digits of accuracy.

Loop Unrolling using –Munroll

This optimization unrolls loops, executing multiple instances of the loop during each iteration. This reduces branch overhead, and can improve execution speed by creating better opportunities for instruction scheduling. A loop with a constant count may be completely unrolled or partially unrolled. A loop with a non-constant count may also be unrolled. A candidate loop must be an innermost loop containing one to four blocks of code. The following shows the use of the `-Munroll` option:

```
$ pgf95 -Munroll prog.f
```

The `-Munroll` option is included as part of `-fast` on all x86 and x64 targets. The loop unroller expands the contents of a loop and reduces the number of times a loop is executed. Branching overhead is reduced when a loop is unrolled two or more times, since each iteration of the unrolled loop corresponds to two or more iterations of the original loop; the number of branch instructions executed is proportionately reduced. When a loop is unrolled completely, the loop's branch overhead is eliminated altogether.

Loop unrolling may be beneficial for the instruction scheduler. When a loop is completely unrolled or unrolled two or more times, opportunities for improved scheduling may be presented. The code generator can take advantage of more possibilities for instruction grouping or filling instruction delays found within the loop.

Example 4-1 and Example 4-2 show the effect of code unrolling on a segment that computes a dot product.

Example 3.1. Dot Product Code

```
REAL*4 A(100), B(100), Z
INTEGER I
DO I=1, 100
  Z = Z + A(i) * B(i)
END DO
END
```

Example 3.2. Unrolled Dot Product Code

```
REAL*4 A(100), B(100), Z
 INTEGER I
 DO I=1, 100, 2
   Z = Z + A(i) * B(i)
   Z = Z + A(i+1) * B(i+1)
 END DO
END
```

Using the `-Minfo` option, the compiler informs you when a loop is being unrolled. For example, a message similar to the following, indicating the line number, and the number of times the code is unrolled, displays when a loop is unrolled:

```
dot:
 5, Loop unrolled 5 times
```

Using the c:<m> and n:<m> sub-options to `-Munroll`, or using `-Mnounroll`, you can control whether and how loops are unrolled on a file-by-file basis. Using directives or pragmas as specified in Chapter 8, "*Using Directives and Pragmas*", you can precisely control whether and how a given loop is unrolled. For a detailed description of the `-Munroll` option, refer to Chapter 2, "*Using Command Line Options*".

Vectorization using –Mvect

The `-Mvect` option is included as part of `-fast` on all x86 and x64 targets. If your program contains computationally-intensive loops, the `-Mvect` option may be helpful. If in addition you specify `-Minfo`, and your code contains loops that can be vectorized, the compiler reports relevant information on the optimizations applied.

When a PGI compiler command is invoked with the `-Mvect` option, the vectorizer scans code searching for loops that are candidates for high-level transformations such as loop distribution, loop interchange, cache tiling, and idiom recognition (replacement of a recognizable code sequence, such as a reduction loop, with optimized code sequences or function calls). When the vectorizer finds vectorization opportunities, it internally rearranges or replaces sections of loops (the vectorizer changes the code generated; your source code's loops are not altered). In addition to performing these loop transformations, the vectorizer produces extensive data dependence information for use by other phases of compilation and detects opportunities to use vector or packed Streaming SIMD Extensions (SSE) instructions on processors where these are supported.

The `-Mvect` option can speed up code which contains well-behaved countable loops which operate on large REAL, REAL*4, REAL*8, INTEGER*4, COMPLEX or COMPLEX DOUBLE arrays in Fortran and their C/C++ counterparts. However, it is possible that some codes will show a decrease in performance when compiled with the `-Mvect` option due to the generation of conditionally executed code segments, inability to determine data alignment, and other code generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled with this option enabled.

Vectorization Sub-options

The vectorizer performs high-level loop transformations on countable loops. A loop is countable if the number of iterations is set only before loop execution and cannot be modified during loop execution. Some of the vectorizer transformations can be controlled by arguments to the `-Mvect` command line option. The

following sections describe the arguments that affect the operation of the vectorizer. In addition, some of these vectorizer operations can be controlled from within code using directives and pragmas. For details on the use of directives and pragmas, refer to Chapter 8, "*Using Directives and Pragmas*," on page 101.

The vectorizer performs the following operations:

- Loop interchange
- Loop splitting
- Loop fusion
- Memory-hierarchy (cache tiling) optimizations
- Generation of SSE instructions on processors where these are supported
- Generation of prefetch instructions on processors where these are supported
- Loop iteration peeling to maximize vector alignment
- Alternate code generation

By default, `-Mvect` without any sub-options is equivalent to:

```
-Mvect=assoc,cachesize=c
```

where c is the actual cache size of the machine.

This enables the options for nested loop transformation and various other vectorizer options. These defaults may vary depending on the target system.

Assoc Option

The option `-Mvect=assoc` instructs the vectorizer to perform associativity conversions that can change the results of a computation due to a round-off error (`-Mvect=noassoc` disables this option). For example, a typical optimization is to change one arithmetic operation to another arithmetic operation that is mathematically correct, but can be computationally different and generate faster code. This option is provided to enable or disable this transformation, since a round-off error for such associativity conversions may produce unacceptable results.

Cachesize Option

The option `-Mvect=cachesize:n` instructs the vectorizer to tile nested loop operations assuming a data cache size of n bytes. By default, the vectorizer attempts to tile nested loop operations, such as matrix multiply, using multi-dimensional strip-mining techniques to maximize re-use of items in the data cache.

SSE Option

The option `-Mvect=sse` instructs the vectorizer to automatically generate packed SSE (Streaming SIMD Extensions), SSE2, and prefetch instructions when vectorizable loops are encountered. SSE instructions, first introduced on Pentium III and AthlonXP processors, operate on single-precision floating-point data, and hence apply only to vectorizable loops that operate on single-precision floating-point data. SSE2 instructions, first introduced on Pentium 4, Xeon and Opteron processors, operate on double-precision floating-point data. Prefetch instructions, first introduced on Pentium III and AthlonXP processors, can be used to improve the

performance of vectorizable loops that operate on either 32-bit or 64-bit floating-point data. Refer to the PGI Release Notes for a concise list of processors that support SSE, SSE2 and prefetch instructions.

Note

Program units compiled with `–Mvect=sse` will not execute on Pentium, Pentium Pro, Pentium II or first generation AMD Athlon processors. They will only execute correctly on Pentium III, Pentium 4, Xeon, EM64T, AthlonXP, Athlon64 and Opteron systems running an SSE-enabled operating system.

Prefetch Option

The option `–Mvect=prefetch` instructs the vectorizer to automatically generate prefetch instructions when vectorizable loops are encountered, even in cases where SSE or SSE2 instructions are not generated. Usually, explicit prefetching is not necessary on Pentium 4, Xeon and Opteron because these processors support hardware prefetching; nonetheless, it sometimes can be worthwhile to experiment with explicit prefetching. Prefetching can be controlled on a loop-by-loop level using prefetch directives, which are described in detail in "Prefetch Directives and Pragmas," on page 107.

Note

Program units compiled with `–Mvect=prefetch` will not execute correctly on Pentium, Pentium Pro, or Pentium II processors. They will execute correctly only on Pentium III, Pentium 4, Xeon, EM64T, AthlonXP, Athlon64 or Opteron systems. In addition, the `prefetch` instruction is only supported on AthlonXP, Athlon64 or Opteron systems and can cause instruction faults on non-AMD processors. For this reason, the PGI compilers do not generate `prefetch` instructions by default on any target.

In addition to these sub-options to `–Mvect`, several other sub-options are supported. Refer to the description of `–M[no]vect` in Chapter 16, "*Command-Line Options Reference*" for a detailed description of all available sub-options.

Vectorization Example Using SSE/SSE2 Instructions

One of the most important vectorization options is `-Mvect=sse`. When you use this option, the compiler automatically generates SSE and SSE2 instructions, where possible, when targeting processors on which these instructions are supported. This process can improve performance by up to a factor of two compared with the equivalent scalar code. All of the PGI Fortran, C and C++ compilers support this capability. The PGI Release Notes show which x86 and x64 processors support these instructions.

Prior to release 7.0, `-Mvect=sse` was omitted from the compiler switch `-fast` but was included in the switch `-fastsse`. Since release 7.0 , `-fast` is synonymous with `-fastsse`; therefore, both options include `-Mvect=sse`.

In the program in Example 3.3, "Vector operation using SSE instructions", the vectorizer recognizes the vector operation in subroutine 'loop' when either the compiler switch `-Mvect=sse` or `-fast` is used. This example shows the compilation, informational messages, and run-time results using the SSE instructions on an AMD Opteron processor-based system, along with issues that affect SSE performance.

First note that the arrays in Example 3.3 are single-precision and that the vector operation is done using a unit stride loop. Thus, this loop can potentially be vectorized using SSE instructions on any processor that

supports SSE or SSE2 instructions. SSE operations can be used to operate on pairs of single-precision floating-point numbers, and do not apply to double-precision floating-point numbers. SSE2 instructions can be used to operate on quads of single-precision floating-point numbers or on pairs of double-precision floating-point numbers.

Loops vectorized using SSE or SSE2 instructions operate much more efficiently when processing vectors that are aligned to a cache-line boundary. You can cause unconstrained data objects of size 16 bytes or greater to be cache-aligned by compiling with the `-Mcache_align` switch. An unconstrained data object is a data object that is not a common block member and not a member of an aggregate data structure.

Note

For stack-based local variables to be properly aligned, the main program or function must be compiled with `-Mcache_align`.

The `-Mcache_align` switch has no effect on the alignment of Fortran allocatable or automatic arrays. If you have arrays that are constrained, such as vectors that are members of Fortran common blocks, you must specifically pad your data structures to ensure proper cache alignment. You can use `-Mcache_align` for only the beginning address of each common block to be cache-aligned.

The following examples show the results of compiling the sample code in Example 3.3 both with and without the option `-Mvect=sse`.

Example 3.3. Vector operation using SSE instructions

```
program vector_op
 parameter (N = 9999)
 real*4 x(N), y(N), z(N), W(N)
 do i = 1, n
   y(i) = i
   z(i) = 2*i
   w(i) = 4*i
 enddo
 do j = 1, 200000
   call loop(x,y,z,w,1.0e0,N)
 enddo
 print *, x(1),x(771),x(3618),x(6498),x(9999)
end
```

```
subroutine loop(a,b,c,d,s,n)
 integer i, n
 real*4 a(n), b(n), c(n), d(n),s
 do i = 1, n
   a(i) = b(i) + c(i) - s * d(i)
 enddo
end
```

Assume the preceding program is compiled as follows, where `-Mvect=nosse` disables SSE vectorization:

```
 % pgf95 -fast -Mvect=nosse -Minfo vadd.f
 vector_op:
 4, Loop unrolled 4 times
 loop:
 18, Loop unrolled 4 times
```

The following output shows a sample result if the generated executable is run and timed on a standalone AMD Opteron 2.2 Ghz system:

```
% /bin/time vadd
-1.000000 -771.000 -3618.000 -6498.00 -9999.00
5.39user 0.00system 0:05.40elapsed 99%CP
```

Now, recompile with SSE vectorization enabled, and you see results similar to these:

```
% pgf95 -fast -Minfo vadd.f -o vadd
vector_op:
4, Unrolled inner loop 8 times
Loop unrolled 7 times (completely unrolled)
loop:
18, Generated 4 alternate loops for the inner loop
Generated vector sse code for inner loop
Generated 3 prefetch instructions for this loop
```

Notice the informational message for the loop at line 18.

- The first two lines of the message indicate that the loop was vectorized, SSE instructions were generated, and four alternate versions of the loop were also generated. The loop count and alignments of the arrays determine which of these versions is executed.
- The last line of the informational message indicates that prefetch instructions have been generated for three loads to minimize latency of data transfers from main memory.

Executing again, you should see results similar to the following:

```
% /bin/time vadd
-1.000000 -771.000 -3618.00 -6498.00
-9999.0
3.59user 0.00system 0:03.59elapsed 100%CPU
```

The result is a 50% speed-up over the equivalent scalar, that is, the non-SSE, version of the program.

Speed-up realized by a given loop or program can vary widely based on a number of factors:

- When the vectors of data are resident in the data cache, performance improvement using vector SSE or SSE2 instructions is most effective.
- If data is aligned properly, performance will be better in general than when using vector SSE operations on unaligned data.
- If the compiler can guarantee that data is aligned properly, even more efficient sequences of SSE instructions can be generated.
- The efficiency of loops that operate on single-precision data can be higher. SSE2 vector instructions can operate on four single-precision elements concurrently, but only two double-precision elements.

Note

Compiling with `–Mvect=sse` can result in numerical differences from the executables generated with less optimization. Certain vectorizable operations, for example dot products, are sensitive to order of operations and the associative transformations necessary to enable vectorization (or parallelization).

Auto-Parallelization using -Mconcur

With the `-Mconcur` option the compiler scans code searching for loops that are candidates for auto-parallelization. `-Mconcur` must be used at both compile-time and link-time. When the parallelizer finds opportunities for auto-parallelization, it parallelizes loops and you are informed of the line or loop being parallelized if the `-Minfo` option is present on the compile line. See "Optimization Controls," on page 264, for a complete specification of `-Mconcur`.

A loop is considered parallelizable if doesn't contain any cross-iteration data dependencies. Cross-iteration dependencies from reductions and expandable scalars are excluded from consideration, enabling more loops to be parallelizable. In general, loops with calls are not parallelized due to unknown side effects. Also, loops with low trip counts are not parallelized since the overhead in setting up and starting a parallel loop will likely outweigh the potential benefits. In addition, the default is to not parallelize innermost loops, since these often by definition are vectorizable using SSE instructions and it is seldom profitable to both vectorize and parallelize the same loop, especially on multi-core processors. Compiler switches and directives are available to let you override most of these restrictions on auto-parallelization.

Auto-parallelization Sub-options

The parallelizer performs various operations that can be controlled by arguments to the `–Mconcur` command line option. The following sections describe these arguments that affect the operation of the vectorizer. In addition, these vectorizer operations can be controlled from within code using directives and pragmas. For details on the use of directives and pragmas, refer to Chapter 8, "*Using Directives and Pragmas*".

By default, `–Mconcur` without any sub-options is equivalent to:

```
-Mconcur=dist:block
```

This enables parallelization of loops with blocked iteration allocation across the available threads of execution. These defaults may vary depending on the target system.

Altcode Option

The option `–Mconcur=altcode` instructs the parallelizer to generate alternate serial code for parallelized loops. If altcode is specified without arguments, the parallelizer determines an appropriate cutoff length and generates serial code to be executed whenever the loop count is less than or equal to that length. If `altcode:n` is specified, the serial altcode is executed whenever the loop count is less than or equal to n. If `noaltcode` is specified, no alternate serial code is generated.

Dist Option

The option `–Mconcur=dist:{block|cyclic}` option specifies whether to assign loop iterations to the available threads in blocks or in a cyclic (round-robin) fashion. Block distribution is the default. If cyclic is specified, iterations are allocated to processors cyclically. That is, processor 0 performs iterations 0, 3, 6, etc.; processor 1 performs iterations 1, 4, 7, etc.; and processor 2 performs iterations 2, 5, 8, etc.

Cncall Option

The option `–Mconcur=cncall` specifies that it is safe to parallelize loops that contain subroutine or function calls. By default, such loops are excluded from consideration for auto-parallelization. Also, no minimum loop

count threshold must be satisfied before parallelization will occur, and last values of scalars are assumed to be safe.

The environment variable NCPUS is checked at run-time for a parallel program. If NCPUS is set to 1, a parallel program runs serially, but will use the parallel routines generated during compilation. If NCPUS is set to a value greater than 1, the specified number of processors will be used to execute the program. Setting NCPUS to a value exceeding the number of physical processors can produce inefficient execution. Executing a program on multiple processors in an environment where some of the processors are being time-shared with another executing job can also result in inefficient execution.

As with the vectorizer, the `-Mconcur` option can speed up code if it contains well-behaved countable loops and/or computationally intensive nested loops that operate on arrays. However, it is possible that some codes will show a decrease in performance on multi-processor systems when compiled with `-Mconcur` due to parallelization overheads, memory bandwidth limitations in the target system, false-sharing of cache lines, or other architectural or code-generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled using this option.

If the compiler is not able to successfully auto-parallelize your application, you should refer to Chapter 5, "*Using OpenMP*". It is possible that insertion of explicit parallelization directives or pragmas, and use of the compiler option `–mp` might enable the application to run in parallel.

Loops That Fail to Parallelize

In spite of the sophisticated analysis and transformations performed by the compiler, programmers may notice loops that are seemingly parallel, but are not parallelized. In this subsection, we look at some examples of common situations where parallelization does not occur.

Innermost Loops

As noted earlier in this chapter, the PGI compilers will not parallelize innermost loops by default, because it is usually not profitable. However, you can override this default using the command-line option `–Mconcur=innermost`.

Timing Loops

Often, loops occur in programs that are similar to timing loops. The outer loop in the following example is one such loop.

```
do j = 1, 2
 do i = 1, n
 a(i) = b(i) + c(i)
1 enddo
enddo
```

The outer loop in the preceding example is not parallelized because the compiler detects a cross-iteration dependence in the assignment to `a(i)`. Suppose the outer loop were parallelized. Then both processors would simultaneously attempt to make assignments into `a(1:n)`. Now in general the values computed by each processor for `a(1:n)` will differ, so that simultaneous assignment into `a(1:n)` will produce values different from sequential execution of the loops.

In this example, values computed for `a(1:n)` don't depend on `j`, so that simultaneous assignment by both processors will not yield incorrect results. However, it is beyond the scope of the compilers' dependence

analysis to determine that values computed in one iteration of a loop don't differ from values computed in another iteration. So the worst case is assumed, and different iterations of the outer loop are assumed to compute different values for `a(1:n)`. Is this assumption too pessimistic? If `j` doesn't occur anywhere within a loop, the loop exists only to cause some delay, most probably to improve timing resolution. It is not usually valid to parallelize timing loops; to do so would distort the timing information for the inner loops.

Scalars

Quite often, scalars will inhibit parallelization of non-innermost loops. There are two separate cases that present problems. In the first case, scalars appear to be expandable, but appear in non-innermost loops, as in the following example.

```
do j = 1, n
 x = b(j)
 do i = 1, n
 a(i,j) = x + c(i,j)
 enddo
enddo
```

There are a number of technical problems to be resolved in order to recognize expandable scalars in non-innermost loops. Until this generalization occurs, scalars like `x` in the preceding code segment inhibit parallelization of loops in which they are assigned. In the following example, scalar `k` is not expandable, and it is not an accumulator for a reduction.

```
 k = 1
 do i = 1, n
 do j = 1, n
1 a(j,i) = b(k) * x
 enddo
 k = i
2 if (i .gt. n/2) k = n - (i - n/2)
 enddo
```

If the outer loop is parallelized, conflicting values are stored into `k` by the various processors. The variable `k` cannot be made local to each processor because its value must remain coherent among the processors. It is possible the loop could be parallelized if all assignments to `k` are placed in critical sections. However, it is not clear where critical sections should be introduced because in general the value for `k` could depend on another scalar (or on `k` itself), and code to obtain the value of other scalars must reside in the same critical section.

In the previous example, the assignment to `k` within a conditional at label 2 prevents `k` from being recognized as an induction variable. If the conditional statement at label 2 is removed, `k` would be an induction variable whose value varies linearly with `j`, and the loop could be parallelized.

Scalar Last Values

During parallelization, scalars within loops often need to be privatized; that is, each execution thread has its own independent copy of the scalar. Problems can arise if a privatized scalar is accessed outside the loop. For example, consider the following loop:

```
for (i = 1; i<N; i++){
 if( f(x[i]) > 5.0 ) t = x[i];
}
 v = t;
```

The value of `t` may not be computed on the last iteration of the loop. Normally, if a scalar is assigned within a loop and used following the loop, the PGI compilers save the last value of the scalar. However, if the loop is parallelized and the scalar is not assigned on every iteration, it may be difficult, without resorting to costly critical sections, to determine on what iteration `t` is last assigned. Analysis allows the compiler to determine that a scalar is assigned on each iteration and hence that the loop is safe to parallelize if the scalar is used later, as illustrated in the following example.

```
for ( i = 1; i < n; i++) {
 if ( x[i] > 0.0 ) {
 t = 2.0;
 }
 else {
 t = 3.0;
 y[i] = ...t;
 }
}
v = t;
```

where `t` is assigned on every iteration of the loop. However, there are cases where a scalar may be privatizable, but if it is used after the loop, it is unsafe to parallelize. Examine the following loop in which each use of `t` within the loop is reached by a definition from the same iteration.

```
for ( i = 1; i < N; i++ ){
 if( x[i] > 0.0 ){
 t = x[i];
 ...
 ...
 y[i] = ...t;
 }
 }
 v = t;
```

Here `t` is privatizable, but the use of `t` outside the loop may yield incorrect results, since the compiler may not be able to detect on which iteration of the parallelized loop `t` is last assigned. The compiler detects the previous cases. When a scalar is used after the loop but is not defined on every iteration of the loop, parallelization does not occur.

When the programmer knows that the scalar is assigned on the last iteration of the loop, the programmer may use a directive or pragma to let the compiler know the loop is safe to parallelize. The Fortran directive `safe_lastval` informs the compiler that, for a given loop, all scalars are assigned in the last iteration of the loop; thus, it is safe to parallelize the loop. We could add the following line to any of our previous examples.

```
cpgi$l safe_lastval
```

The resulting code looks similar to this:

```
cpgi$l safe_lastval
...
for (i = 1; i<N; i++){
 if( f(x[i]) > 5.0 ) t = x[i];
}
 v = t;
```

In addition, a command-line option `-M`safe_lastval, provides this information for all loops within the routines being compiled, which essentially provides global scope.

Processor-Specific Optimization & the Unified Binary

Different processors have differences, some subtle, in hardware features such as instruction sets and cache size. The compilers make architecture-specific decisions about things such as instruction selection, instruction scheduling, and vectorization. By default, the PGI compilers produce code specifically targeted to the type of processor on which the compilation is performed. That is, the default is to use all supported instructions wherever possible when compiling on a given system. As a result, executables created on a given system may not be usable on previous generation systems. For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.

All PGI compilers have the capability of generating *unified binaries*, which provide a low-overhead means for generating a single executable that is compatible with and has good performance on more than one hardware platform.

You can use the `-tp` option to control compilation behavior by specifying the processor or processors with which the generated code is compatible. The compilers generate and combine into one executable multiple binary code streams, each optimized for a specific platform. At run-time, the one executable senses the environment and dynamically selects the appropriate code stream. For specific information on the `-tp` option, refer to –tp <target> [,target...].

Executable size is automatically controlled via unified binary culling. Only those functions and subroutines where the target affects the generated code have unique binary images, resulting in a code-size savings of from 10% to 90% compared to generating full copies of code for each target.

Programs can use the PGI Unified Binary even if all of the object files and libraries are not compiled as unified binaries. Like any other object file, you can use PGI Unified Binary object files to create programs or libraries. No special start up code is needed; support is linked in from the PGI libraries.

The `-Mipa` option disables generation of PGI Unified Binary. Instead, the default target auto-detect rules for the host are used to select the target processor.

Interprocedural Analysis and Optimization using –Mipa

The PGI Fortran, C and C++ compilers use interprocedural analysis (IPA) that results in minimal changes to makefiles and the standard edit-build-run application development cycle. Other than adding `-Mipa` to the command line, no other changes are required. For reference and background, the process of building a program without IPA is described later in this section, followed by the minor modifications required to use IPA with the PGI compilers. While the PGCC compiler is used here to show how IPA works, similar capabilities apply to each of the PGI Fortran, C and C++ compilers.

Note

The examples use Linux file naming conventions. On Windows, '.o' files would be '.obj' files, and 'a.out' files would be '.exe' files.

Building a Program Without IPA – Single Step

Using the pgcc command-level compiler driver, multiple source files can be compiled and linked into a single executable with one command. The following example compiles and links three source files:

```
% pgcc -o a.out file1.c file2.c file3.c
```

In actuality, the pgcc driver executes several steps to produce the assembly code and object files corresponding to each source file, and subsequently to link the object files together into a single executable file. This command is roughly equivalent to the following commands performed individually:

```
% pgcc -S -o file1.s file1.c
% as -o file1.o file1.s
% pgcc -S -o file2.s file2.c
% as -o file2.o file2.s
% pgcc -S -o file3.s file3.c
% as -o file3.o file3.s
% pgcc -o a.out file1.o file2.o file3.o
```

If any of the three source files is edited, the executable can be rebuilt with the same command line:

```
% pgcc -o a.out file1.c file2.c file3.c
```

This always works as intended, but has the side-effect of recompiling all of the source files, even if only one has changed. For applications with a large number of source files, this can be time-consuming and inefficient.

Building a Program Without IPA - Several Steps

It is also possible to use individual pgcc commands to compile each source file into a corresponding object file, and one to link the resulting object files into an executable:

```
% pgcc -c file1.c
% pgcc -c file2.c
% pgcc -c file3.c
% pgcc -o a.out file1.o file2.o file3.o
```

The pgcc driver invokes the compiler and assembler as required to process each source file, and invokes the linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% pgcc -c file1.c
% pgcc -o a.out file1.o file2.o file3.o
```

Building a Program Without IPA Using Make

The program compilation and linking process can be simplified greatly using the `make` utility on systems where it is supported. Suppose you create a `makefile` containing the following lines:

```
a.out: file1.o file2.o file3.o
 pgcc $(OPT) -o a.out file1.o file2.o file3.o
file1.o: file1.c
 pgcc $(OPT) -c file1.c
file2.o: file2.c
 pgcc $(OPT) -c file2.c
file3.o: file3.c
 pgcc $(OPT) -c file3.c
```

It is then possible to type a single make command:

```
% make
```

The `make` utility determines which object files are out of date with respect to their corresponding source files, and invokes the compiler to recompile only those source files and to relink the executable. If you subsequently

edit one or more source files, the executable can be rebuilt with the minimum number of recompilations using the same single `make` command.

Building a Program with IPA

Interprocedural analysis and optimization (IPA) by the PGI compilers alters the standard and `make` utility command-level interfaces as little as possible. IPA occurs in three phases:

- **Collection**: Create a summary of each function or procedure, collecting the useful information for interprocedural optimizations. This is done during the compile step if the `-Mipa` switch is present on the command line; summary information is collected and stored in the object file.
- **Propagation**: Process all the object files to propagate the interprocedural summary information across function and file boundaries. This is done during the link step, when all the object files are combined, if the `-Mipa` switch is present on the link command line.
- **Recompile/Optimization**: Recompile each of the object files with the propagated interprocedural information, producing a specialized object file. This process is also performed during the link step when the `-Mipa` switch is present on the link command line.

When linking with `-Mipa`, the PGI compilers automatically regenerate IPA-optimized versions of each object file, essentially recompiling each file. If there are IPA-optimized objects from a previous build, the compilers will minimize the recompile time by reusing those objects if they are still valid. They will still be valid if the IPA-optimized object is newer than the original object file, and the propagated IPA information for that file has not changed since it was optimized.

After each object file has been recompiled, the regular linker is invoked to build the application with the IPA-optimized object files. The IPA-optimized object files are saved in the same directory as the original object files, for use in subsequent program builds.

Building a Program with IPA - Single Step

By adding the `-Mipa` command line switch, several source files can be compiled and linked with interprocedural optimizations with one command:

```
% pgcc -Mipa=fast -o a.out file1.c file2.c file3.c
```

Just like compiling without `-Mipa`, the driver executes several steps to produce the assembly and object files to create the executable:

```
% pgcc -Mipa=fast -S -o file1.s file1.c
% as -o file1.o file1.s
% pgcc -Mipa=fast -S -o file2.s file2.c
% as -o file2.o file2.s
% pgcc -Mipa=fast -S -o file3.s file3.c
% as -o file3.o file3.s
% pgcc -Mipa=fast -o a.out file1.o file2.o file3.o
```

In the last step, an IPA linker is invoked to read all the IPA summary information and perform the interprocedural propagation. The IPA linker reinvokes the compiler on each of the object files to recompile them with interprocedural information. This creates three new objects with mangled names:

```
file1_ipa5_a.out.oo.o, file2_ipa5_a.out.oo.o, file2_ipa5_a.out.oo.o
```

The system linker is then invoked to link these IPA-optimized objects into the final executable. Later, if one of the three source files is edited, the executable can be rebuilt with the same command line:

```
% pgcc -Mipa=fast -o a.out file1.c file2.c file3.c
```

This will work, but again has the side-effect of compiling each source file, and recompiling each object file at link time.

Building a Program with IPA - Several Steps

Just by adding the `–Mipa` command-line switch, it is possible to use individual pgcc commands to compile each source file, followed by a command to link the resulting object files into an executable:

```
% pgcc -Mipa=fast -c file1.c
% pgcc -Mipa=fast -c file2.c
% pgcc -Mipa=fast -c file3.c
% pgcc -Mipa=fast -o a.out file1.o file2.o file3.o
```

The pgcc driver invokes the compiler and assembler as required to process each source file, and invokes the IPA linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% pgcc -Mipa=fast -c file1.c
% pgcc -Mipa=fast -o a.out file1.o file2.o file3.o
```

When the IPA linker is invoked, it will determine that the IPA-optimized object for `file1.o` (`file1_ipa5_a.out.oo.o`) is stale, since it is older than the object file1.o, and hence will need to be rebuilt, and will reinvoke the compiler to generate it. In addition, depending on the nature of the changes to the source file file1.c, the interprocedural optimizations previously performed for file2 and file3 may now be inaccurate. For instance, IPA may have propagated a constant argument value in a call from a function in `file1.c` to a function in `file2.c`; if the value of the argument has changed, any optimizations based on that constant value are invalid. The IPA linker will determine which, if any, of any previously created IPA-optimized objects need to be regenerated, and will reinvoke the compiler as appropriate to regenerate them. Only those objects that are stale or which have new or different IPA information will be regenerated, which saves on compile time.

Building a Program with IPA Using Make

As in the previous two sections, programs can be built with IPA using the make utility. Just add the command-line switch `–Mipa`, as shown here:

```
OPT=-Mipa=fast a.out: file1.o file2.o file3.o
 pgcc $(OPT) -o a.out file1.o file2.o file3.o
file1.o: file1.c
 pgcc $(OPT) -c file1.c
file2.o: file2.c
 pgcc $(OPT) -c file2.c
file3.o: file3.c
 pgcc $(OPT) -c file3.c
```

Using the single `make` command invokes the compiler to generate any object files that are out-of-date, then invokes pgcc to link the objects into the executable; at link time, pgcc calls the IPA linker to regenerate any stale or invalid IPA-optimized objects.

```
% make
```

Questions about IPA

1. Why is the object file so large?

   An object file created with `-Mipa` contains several additional sections. One is the summary information used to drive the interprocedural analysis. In addition, the object file contains the compiler internal representation of the source file, so the file can be recompiled at link time with interprocedural optimizations. There may be additional information when inlining is enabled. The total size of the object file may be 5-10 times its original size. The extra sections are not added to the final executable.

2. What if I compile with `-Mipa` and link without `-Mipa`?

   The PGI compilers generate a legal object file, even when the source file is compiled with `-Mipa`. If you compile with `-Mipa` and link without `-Mipa`, the linker is invoked on the original object files. A legal executable will be generated; while this will not have the benefit of interprocedural optimizations, any other optimizations will apply.

3. What if I compile without `-Mipa` and link with `-Mipa`?

   At link time, the IPA linker must have summary information about all the functions or routines used in the program. This information is created only when a file is compiled with `-Mipa`. If you compile a file without `-Mipa` and then try to get interprocedural optimizations by linking with `-Mipa`, the IPA linker will issue a message that some routines have no IPA summary information, and will proceed to run the system linker using the original object files. If some files were compiled with `-Mipa` and others were not, it will determine the safest approximation of the IPA summary information for those files not compiled with `-Mipa`, and use that to recompile the other files using interprocedural optimizations.

4. Can I build multiple applications in the same directory with `-Mipa`?

   Yes. Suppose you have three source files: `main1.c`, `main2.c`, and `sub.c`, where `sub.c` is shared between the two applications. Suppose you build the first application with `-Mipa`, using this command:

   ```
   % pgcc -Mipa=fast -o app1 main1.c sub.c
   ```

   The the IPA linker creates two IPA-optimized object files:

   ```
   main1_ipa4_app1.o sub_ipa4_app1.oo
   ```

   It uses them to build the first application. Now suppose you build the second application using this command:

   ```
   % pgcc -Mipa=fast -o app2 main2.c sub.c
   ```

   The IPA linker creates two more IPA-optimized object files:

   ```
   main2_ipa4_app2.oo sub_ipa4_app2.oo
   ```

   ### Note

   There are now three object files for `sub.c`: the original `sub.o`, and two IPA-optimized objects, one for each application in which it appears.

5. How is the mangled name for the IPA-optimized object files generated?

   The mangled name has '_ipa' appended, followed by the decimal number of the length of the executable file name, followed by an underscore and the executable file name itself. The suffix is changed to .oo (on Linux or Mac OS X) or .oobj (on Windows) so linking *.o or *.obj does not pull in the IPA-optimized objects. If the IPA linker determines that the file would not benefit from any interprocedural optimizations, it does not have to recompile the file at link time and uses the original object.

Profile-Feedback Optimization using –Mpfi/–Mpfo

The PGI compilers support many common profile-feedback optimizations, including semi-invariant value optimizations and block placement. These are performed under control of the `-Mpfi`/`-Mpfo` command-line options.

When invoked with the `-Mpfi` option, the PGI compilers instrument the generated executable for collection of profile and data feedback information. This information can be used in subsequent compilations that include the `-Mpfo` optimization option. `-Mpfi` must be used at both compile-time and link-time. Programs compiled with `-Mpfi` include extra code to collect run-time statistics and write them out to a trace file. When the resulting program is executed, a profile feedback trace file `pgfi.out` is generated in the current working directory.

Note

Programs compiled and linked with `-Mpfi` execute more slowly due to the instrumentation and data collection overhead. You should use executables compiled with `-Mpfi` only for execution of training runs.

When invoked with the `-Mpfo` option, the PGI compilers use data from a `pgfi.out` profile feedback tracefile to enable or enhance certain performance optimizations. Use of this option requires the presence of a `pgfi.out` trace file in the current working directory.

Default Optimization Levels

The following table shows the interaction between the –O<level> , –g, and `-M<opt>` options. In the table, level can be 0, 1, 2, 3 or 4, and <opt> can be `vect`, `concur`, `unroll` or `ipa`. The default optimization level is dependent upon these command-line options.

Table 3.1. Optimization and –O, –g and –M<opt> Options

Optimize Option	Debug Option	–M<opt> Option	Optimization Level
none	none	none	1
none	none	–M<opt>	2
none	–g	none	0
–O	none or –g	none	2

Optimize Option	Debug Option	–M<opt> Option	Optimization Level
–Olevel	none or –g	none	level
–Olevel <= 2	none or –g	–M<opt>	2

Code that is not optimized yet compiled using the option `-O0` can be significantly slower than code generated at other optimization levels. The `-M<opt>` option, where <opt> is `vect`, `concur`, `unroll` or `ipa`, sets the optimization level to 2 if no `-O` options are supplied. Both the `-fast` and the `-fastsse` options set the optimization level to a target-dependent optimization level if no `-O` options are supplied.

Local Optimization Using Directives and Pragmas

Command-line options let you specify optimizations for an entire source file. Directives supplied within a Fortran source file and pragmas supplied within a C or C++ source file provide information to the compiler and alter the effects of certain command-line options or the default behavior of the compiler. (Many directives have a corresponding command-line option.)

While a command line option affects the entire source file that is being compiled, directives and pragmas let you do the following:

- Apply, or disable, the effects of a particular command-line option to selected subprograms or to selected loops in the source file (for example, an optimization).
- Globally override command-line options.
- Tune selected routines or loops based on your knowledge or on information obtained through profiling.

Chapter 8, "*Using Directives and Pragmas*" provides details on how to add directives and pragmas to your source files.

Execution Timing and Instruction Counting

As this chapter describes, once you have a program that compiles, executes and gives correct results, you may optimize your code for execution efficiency. Selecting the correct optimization level requires some thought and may require that you compare several optimization levels before arriving at the best solution. To compare optimization levels, you need to measure the execution time for your program. There are several approaches you can take for timing execution. You can use shell commands that provide execution time statistics, you can include function calls in your code that provide timing information, or you can profile sections of code. Timing functions available with the PGI compilers include 3F timing routines, the SECNDS pre-declared function in PGF77 or PGF95, or the SYSTEM_CLOCK or CPU_CLOCK intrinsics in PGF95 or PGHPF. In general, when timing a program, you should try to eliminate or reduce the amount of system level activities such as I/O, program loading, and task switching.

The following example shows a fragment that indicates how to use SYSTEM_CLOCK effectively within an F90, F95 or HPF program unit.

Example 3.4. Using SYSTEM_CLOCK code fragment

```
 . . .
 integer :: nprocs, hz, clock0, clock1
 real :: time
 integer, allocatable :: t(:)
!hpf$ distribute t(cyclic)
#if defined (HPF)
 allocate (t(number_of_processors()))
#elif defined (_OPENMP)
 allocate (t(OMP_GET_NUM_THREADS()))
#else
 allocate (t(1))
#endif
 call system_clock (count_rate=hz)
!
 call system_clock(count=clock0)
 < do work>
 call system_clock(count=clock1)
!
 t = (clock1 - clock0)
 time = real (sum(t)) / (real(hz) * size(t))
 . . .
```

Portability of Multi-Threaded Programs on Linux

PGI has created two libraries - libpgbind and libnuma - to handle the variations between various implementations of Linux.

Some older versions of Linux are lacking certain features that support multi-processor and multi-core systems, in particular, the system call 'sched_setaffinity' and the numa library libnuma. The PGI run-time library uses these features to implement some `–Mconcur` and `–mp` operations.

These variations have led to the creation of two PGI libraries, libpgbind and libnuma. These libraries are used on all 32-bit and 64-bit Linux systems, but are not needed on Windows or Mac OS X.

When a program is linked with the system libnuma library, the program depends on the libnuma library in order to run. On systems without a system libnuma library, the PGI version of libnuma provides the required stubs so that the program links and executes properly.

If the program is linked with libpgbind and libnuma, the differences between systems is masked by the different versions of libpgbind and libnuma. In particular, PGI provides two versions of libpgbind: one for systems with working support for sched_setaffinity and another for systems that do not.

When a program is deployed to the target system, the proper set of libraries, real or stub, should be deployed with the program.

This facility requires that the program be dynamically linked with libpgbind and libnuma.

libpgbind

On some versions of Linux, the system call sched_setaffinity does not exist or does not work. The library libpgbind is used to work around this problem.

During installation, a small test program is compiled, linked, and executed. If the test program compiles, links, and executes successfully, the installed version of libpgbind calls the system sched_setaffinity, otherwise the stub version is installed.

libnuma

Not all systems have libnuma. Typically, only numa systems will have this library. PGI supplies a stub version of libnuma which satisfies the calls from the PGI run-time to libnuma. Note that libnuma is a shared library that is linked dynamically at run-time.

The reason to have a numa library on all systems is to allow multi-threaded programs, such as programs compiled with `-Mconcur` or `-mp` , to be compiled, linked, and executed without regard to whether the host or target systems has a numa library. When the numa library is not available, a multi-threaded program still runs because the calls to the numa library are satisfied by the PGI stub library.

During installation, the installation procedure checks for the existence of a real libnuma among the system libraries. If the real library is not found, the PGI stub version is substituted.

Chapter 4. Using Function Inlining

Function inlining replaces a call to a function or a subroutine with the body of the function or subroutine. This can speed up execution by eliminating parameter passing and function/subroutine call and return overhead. It also allows the compiler to optimize the function with the rest of the code. Note that using function inlining indiscriminately can result in much larger code size and no increase in execution speed.

The PGI compilers provide two categories of inlining:

- **Automatic inlining** - During the compilation process, a hidden pass precedes the compilation pass. This hidden pass extracts functions that are candidates for inlining. The inlining of functions occurs as the source files are compiled.
- **Inline libraries** - You create inline libraries, for example using the pgf95 compiler driver and the `-o` and `-Mextract` options. There is no hidden extract pass but you must ensure that any files that depend on the inline library use the latest version of the inline library.

There are important restrictions on inlining. Inlining only applies to certain types of functions. Refer to "Restrictions on Inlining," on page 53 for more details on function inlining limitations.

This chapter describes how to use the following options related to function inlining:

```
-Mextract
-Minline
-Mrecursive
```

Invoking Function Inlining

To invoke the function inliner, use the `-Minline` option. If you do not specify an inline library, the compiler performs a special prepass on all source files named on the compiler command line before it compiles any of them. This pass extracts functions that meet the requirements for inlining and puts them in a temporary inline library for use by the compilation pass.

Several `-Minline` suboptions let you determine the selection criteria for functions to be inlined. These suboptions include:

except:`func`
: Inlines all eligible functions except `func`, a function in the source text. you can use a comma-separated list to specify multiple functions.

[name:]`func`
: Inlines all functions in the source text whose name matches `func`. you can use a comma-separated list to specify multiple functions.

[size:]`n`
: Inlines functions with a statement count less than or equal to `n`, the specified size.

> **Note**
>
> The size `n` may not exactly equal the number of statements in a selected function; the size parameter is merely a rough gauge.

levels:`n`
: Inlines `n` level of function calling levels. The default number is one (1). Using a level greater than one indicates that function calls within inlined functions may be replaced with inlined code. This approach allows the function inliner to automatically perform a sequence of inline and extract processes.

[lib:]`file.ext`
: Instructs the inliner to inline the functions within the library file `file.ext`. If no inline library is specified, functions are extracted from a temporary library created during an extract prepass.

> **Tip**
>
> Create the library file using the `-Mextract` option.

If you specify both a function name and a size n, the compiler inlines functions that match the function name *or* have n or fewer statements.

If a name is used without a keyword, then a name with a period is assumed to be an inline library and a name without a period is assumed to be a function name. If a number is used without a keyword, the number is assumed to be a size.

In the following example, the compiler inlines functions with fewer than approximately 100 statements in the source file `myprog.f` and writes the executable code in the default output file `a.out`.

```
$ pgf95 -Minline=size:100 myprog.f
```

Refer to "–M Options by Category," on page 253 for more information on the `-Minline` options.

Using an Inline Library

If you specify one or more inline libraries on the command line with the `-Minline` option, the compiler does not perform an initial extract pass. The compiler selects functions to inline from the specified inline library. If you also specify a size or function name, all functions in the inline library meeting the selection criteria are selected for inline expansion at points in the source text where they are called.

If you do not specify a function name or a size limitation for the `-Minline` option, the compiler inlines every function in the inline library that matches a function in the source text.

In the following example, the compiler inlines the function `proc` from the inline library `lib.il` and writes the executable code in the default output file `a.out`.

```
$ pgf95 -Minline=name:proc,lib:lib.il myprog.f
```

The following command line is equivalent to the preceding line, with the exception that in the following example does not use the keywords `name:` and `lib:`. You typically use keywords to avoid name conflicts when you use an inline library name that does not contain a period. Otherwise, without the keywords, a period informs the compiler that the file on the command line is an inline library.

```
$ pgf95 -Minline=proc,lib.il myprog.f
```

Creating an Inline Library

You can create or update an inline library using the `-Mextract` command-line option. If you do not specify selection criteria with the `-Mextract` option, the compiler attempts to extract all subprograms.

Several `-Mextract` options let you determine the selection criteria for creating or updating an inline library. These selection criteria include:

`func`
: Extracts the function `func`. you can use a comma-separated list to specify multiple functions.

[name:]`func`
: Extracts the functions whose name matches `func`, a function in the source text.

[size:]`n`
: Limits the size of the extracted functions to functions with a statement count less than or equal to `n`, the specified size.

> **Note**
>
> The size n may not exactly equal the number of statements in a selected function; the size parameter is merely a rough gauge.

[lib:]`ext.lib`
: Stores the extracted information in the library directory `ext.lib`.

  If no inline library is specified, functions are extracted to a temporary library created during an extract prepass for use during the compilation stage.

When you use the `-Mextract` option, only the extract phase is performed; the compile and link phases are not performed. The output of an extract pass is a library of functions available for inlining. This output is placed in the inline library file specified on the command line with the `–o` filename specification. If the library file exists, new information is appended to it. If the file does not exist, it is created. You can use a command similar to the following:

```
$ pgf95 -Mextract=lib:lib.il myfunc.f
```

You can use the `-Minline` option with the `-Mextract` option. In this case, the extracted library of functions can have other functions inlined into the library. Using both options enables you to obtain more than one level of inlining. In this situation, if you do not specify a library with the `–Minline` option, the inline process

consists of two extract passes. The first pass is a hidden pass implied by the `-Minline` option, during which the compiler extracts functions and places them into a temporary library. The second pass uses the results of the first pass but puts its results into the library that you specify with the `-o` option.

Working with Inline Libraries

An inline library is implemented as a directory with each inline function in the library stored as a file using an encoded form of the inlinable function.

A special file named `TOC` in the inline library directory serves as a table of contents for the inline library. This is a printable, ASCII file which you can examine to locate information about the library contents, such as names and sizes of functions, the source file from which they were extracted, the version number of the extractor which created the entry, and so on.

Libraries and their elements can be manipulated using ordinary system commands.

- Inline libraries can be copied or renamed.
- Elements of libraries can be deleted or copied from one library to another.
- The `ls` or `dir` command can be used to determine the last-change date of a library entry.

Dependencies

When a library is created or updated using one of the PGI compilers, the last-change date of the library directory is updated. This allows a library to be listed as a dependence in a makefile and ensures that the necessary compilations are performed when a library is changed.

Updating Inline Libraries - Makefiles

If you use inline libraries you must be certain that they remain up-to-date with the source files into which they are inlined. One way to assure inline libraries are updated is to include them in a makefile.

The makefile fragment in the following example assumes the file `utils.f` contains a number of small functions used in the files `parser.f` and `alloc.f`.

This portion of the makefile:

- Maintains the inline library `utils.il`.
- Updates the library whenever you change `utils.f` or one of the include files it uses.
- Compiles `parser.f` and `alloc.f` whenever you update the library.

Example 4.1. Sample Makefile

```
SRC = mydir
FC = pgf95
FFLAGS = -O2
main.o: $(SRC)/main.f $(SRC)/global.h
 $(FC) $(FFLAGS) -c $(SRC)/main.f
utils.o: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
 $(FC) $(FFLAGS) -c $(SRC)/utils.f
```

```
utils.il: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
 $(FC) $(FFLAGS) -Mextract=15 -o utils.il utils.f
parser.o: $(SRC)/parser.f $(SRC)/global.h utils.il
 $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/parser.f
alloc.o: $(SRC)/alloc.f $(SRC)/global.h utils.il
 $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/alloc.f
myprog: main.o utils.o parser.o alloc.o
 $(FC) -o myprog main.o utils.o parser.o alloc.o
```

Error Detection during Inlining

You can specify the `–Minfo=inline` option to request inlining information from the compiler when you invoke the inliner. For example:

```
$ pgf95 -Minline=mylib.il -Minfo=inline myext.f
```

Examples

Assume the program `dhry` consists of a single source file `dhry.f`. The following command line builds an executable file for `dhry` in which proc7 is inlined wherever it is called:

```
$ pgf95 dhry.f -Minline=proc7
```

The following command lines build an executable file for `dhry` in which proc7 plus any functions of approximately 10 or fewer statements are inlined (one level only).

> **Note**
>
> The specified functions are inlined only if they are previously placed in the inline library, `temp.il`, during the extract phase.

```
$ pgf95 dhry.f -Mextract=lib:temp.il
$ pgf95 dhry.f -Minline=10,proc7,temp.il
```

Using the same source file `dhry.f`, the following example builds an executable for `dhry` in which all functions of roughly ten or fewer statements are inlined. Two levels of inlining are performed. This means that if function A calls function B, and B calls C, and both B and C are inlinable, then the version of B which is inlined into A will have had C inlined into it.

```
$ pgf95 dhry.f -Minline=size:10,levels:2
```

Restrictions on Inlining

The following Fortran subprograms cannot be extracted:

- Main or BLOCK DATA programs.
- Subprograms containing alternate return, assigned GO TO, DATA, SAVE, or EQUIVALENCE statements.
- Subprograms containing FORMAT statements.
- Subprograms containing multiple entries.

A Fortran subprogram is not inlined if any of the following applies:

- It is referenced in a statement function.
- A common block mismatch exists; in other words, the caller must contain all common blocks specified in the callee, and elements of the common blocks must agree in name, order, and type (except that the caller's common block can have additional members appended to the end of the common block).
- An argument mismatch exists; in other words, the number and type (size) of actual and formal parameters must be equal.
- A name clash exists, such as a call to subroutine `xyz` in the extracted subprogram and a variable named `xyz` in the caller.

The following types of C and C++ functions cannot be inlined:

- Functions containing switch statements
- Functions which reference a static variable whose definition is nested within the function
- Function which accept a variable number of arguments

Certain C/C++ functions can only be inlined into the file that contains their definition:

- Static functions
- Functions which call a static function
- Functions which reference a static variable

Chapter 5. Using OpenMP

The PGF77 and PGF95 Fortran compilers support the OpenMP Fortran Application Program Interface. The PGCC ANSI C and C++ compilers support the OpenMP C/C++ Application Program Interface.

OpenMP is a specification for a set of compiler directives, an applications programming interface (API), and a set of environment variables that can be used to specify shared memory parallelism in FORTRAN and C/C++ programs. OpenMP may be used to obtain most of the parallel performance you can expect from your code, or it may serve as a stepping stone to parallelizing an entire application with MPI.

This chapter provides information on OpenMP as it is supported by PGI compilers.

Use the `-mp` compiler switch to enable processing of the OMP pragmas listed in this chapter. C++ applications will also compile with thread-safe versions of STL header files. Users must link with the `-mp` switch to link the OpenMP runtime library, and for C++, the thread-safe Standard Template Library.

Note

The C++ Standard Template library has been made thread-safe to extent allowed in the STLport code: simultaneous accesses to distinct containers are safe, simultaneous reads to shared containers are also safe. However, simultaneous writes to shared containers must be protected by **#pragma omp critical** sections.

This chapter describes how to use the following options related to using OpenMP:

`-mp`

OpenMP Overview

Let's look at the OpenMP shared-memory parallel programming model and some common OpenMP terminology.

OpenMP Shared-Memory Parallel Programming Model

The OpenMP shared-memory programming model is a collection of compiler directives or pragmas, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran and C/C++ programs.

Fortran directives and C/C++ pragmas
: Allow users to mark sections of code that can be executed in parallel when the code is compiled using the `-mp` switch. When this switch is not present, the compiler ignores these directives and pragmas.

  OpenMP Fortran directives begin with `!$OMP` , `C$OMP`, or `*$OMP`, beginning in column 1. OpenMP pragmas for C/C++ begin with `#pragma omp`. This format allows the user to have a single source for use with or without the `-mp` switch, as these lines are then merely viewed as comments when `-mp` is not present or the compilers are not capable of handling directives or C/C++ pragmas.

  These directives and pragmas allow the user to create task, loop, and parallel section work-sharing constructs and synchronization constructs. They also allow the user to define how data is shared or copied between parallel threads of execution.

  Fortran directives and C/C++ pragmas include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations or C/C++ for loop iterations should be split among the available threads of execution, and synchronization constructs.

  Note

  The data environment is controlled either by using clauses on the directives or pragmas, or with additional directives or pragmas.

Run-time library routines
: Are available to query the parallel run-time environment, for example to determine how many threads are participating in execution of a parallel region.

Environment variables
: Are available to control the execution behavior of parallel programs. For more information on OpenMP, see *www.openmp.org*.

Macro substitution
: C and C++ omp pragmas are subject to macro replacement after `#pragma omp`.

Terminology

For OpenMP 3.0 there are a number of terms for which it is useful to have common definitions.

Thread
: An execution entity with a stack and associated static memory, called *threadprivate memory*.

  - An OpenMP thread is a thread that is managed by the OpenMP runtime system.
  - A thread-safe routine is a routine that performs the intended function even when executed concurrently, that is, by more than one thread.

Region
: All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

Regions are *nested* if one region is (dynamically) enclosed by another region, that is, a region is encountered during the execution of another region. PGI currently does not support nested parallel regions.

Parallel region

In OpenMP 3.0 there is a distinction between a parallel region and an active parallel region. A parallel region can be either inactive or active.

- An inactive parallel region is executed by a single thread.
- An active parallel region is a parallel region that is executed by a team consisting of more than one thread.

Note

The definition of an active parallel region changed between OpenMP 2.5 and OpenMP 3.0. In OpenMP 2.5, the definition was a parallel region whose IF clause evaluates to true. To examine the significance of this change, look at the following example:

```
     program test
     logical omp_in_parallel

!$omp parallel
     print *, omp_in_parallel()
!$omp end parallel

     stop
     end
```

Suppose we run this program with OMP_NUM_THREADS set to one. In OpenMP 2.5, this program yields T while in OpenMP 3.0, the program yields F. In OpenMP 3.0, execution is not occurring by more than one thread. Therefore, change in this definition may mean previous programs require modification.

PGI currently does not support nested parallel regions so currently has only one level of active parallel regions.

Task

A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

OpenMP Example

Look at the following simple OpenMP example involving loops.

Example 5.1. OpenMP Loop Example

```
      PROGRAM MAIN
      INTEGER I, N, OMP_GET_THREAD_NUM
      REAL*8 V(1000), GSUM, LSUM

      GSUM = 0.0D0
      N = 1000
```

```
      DO I = 1, N
         V(I) = DBLE(I)
      ENDDO

!$OMP PARALLEL PRIVATE(I,LSUM) SHARED(V,GSUM,N)
      LSUM = 0.0D0
!$OMP DO
      DO I = 1, N
         LSUM = LSUM + V(I)
      ENDDO
!$OMP END DO
!$OMP CRITICAL
      print *, "Thread ",OMP_GET_THREAD_NUM()," local sum: ",LSUM
      GSUM = GSUM + LSUM
!$OMP END CRITICAL
!$OMP END PARALLEL

      PRINT *, "Global Sum: ",GSUM

      STOP
      END
```

If you execute this example with the environment variable OMP_NUM_THREADS set to 4, then the output looks similar to this:

```
Thread            0 local sum:    31375.00000000000
Thread            1 local sum:    93875.00000000000
Thread            2 local sum:    156375.0000000000
Thread            3 local sum:    218875.0000000000
Global Sum:    500500.0000000000
FORTRAN STOP
```

Task Overview

Every part of an OpenMP program is part of a task. A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

In the following sections, we use this terminology:

Task
: The package of code and instructions for allocating data created when a thread encounters a task construct. A task can be implicit or explicit.

  - An explicit task is a task generated when a task construct is encountered during execution.
  - An implicit task is a task generated by the implicit parallel region or generated when a parallel construct is encountered during execution.

Task construct
: A task directive plus a structured block

Task region
: The dynamic sequence of instructions produced by the execution of a task by a thread.

Fortran Parallelization Directives

Parallelization directives are comments in a program that are interpreted by the PGI Fortran compilers when the option `-mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

With the exception of the SGI-compatible DOACROSS directive, the *sentinel* must comply with these rules:

- Be one of these: !OMP, COMP, or *$OMP.
- Must start in column 1 (one).
- Must appear as a single word without embedded white space.
- The sentinel marking a DOACROSS directive is C$.

The *directive_name* can be any of the directives listed in Table 5.1, "Directive and Pragma Summary Table," on page 61. The valid clauses depend on the directive. Chapter 17, "*OpenMP Reference Information*" provides a list of directives and their clauses, their usage, and examples.

In addition to the sentinel rules, the directive must also comply with these rules:

- Standard Fortran syntax restrictions, such as line length, case insensitivity, and so on, apply to the directive line.
- Initial directive lines must have a space or zero in column six.
- Continuation directive lines must have a character other than a space or a zero in column six. Continuation lines for C$DOACROSS directives are specified using the C$& sentinel.
- Directives which are presented in pairs must be used in pairs.

Clauses associated with directives have these characteristics:

- The order in which clauses appear in the parallelization directives is not significant.
- Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause.
- Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

C/C++ Parallelization Pragmas

Parallelization pragmas are #pragma statements in a C or C++ program that are interpreted by the PGCC C and C++ compilers when the option -mp is specified on the command line. The form of a parallelization pragma is:

```
#pragma omp pragma_name [clauses]
```

The format for pragmas include these standards:

- The pragmas follow the conventions of the C and C++ standards.
- Whitespace can appear before and after the #.
- Preprocessing tokens following the #pragma omp are subject to macro replacement.
- The order in which clauses appear in the parallelization pragmas is not significant.
- Spaces separate clauses within the pragmas.
- Clauses on pragmas may be repeated as needed subject to the restrictions listed in the description of each clause.

For the purposes of the OpenMP pragmas, a C/C++ structured block is defined to be a statement or compound statement (a sequence of statements beginning with { and ending with }) that has a single entry and a single exit. No statement or compound statement is a C/C++ structured block if there is a jump into or out of that statement.

Directive and Pragma Recognition

The compiler option `-mp` enables recognition of the parallelization directives and pragmas. The use of this option also implies:

`-Mreentrant`
: Local variables are placed on the stack and optimizations, such as `-Mnoframe`, that may result in non-reentrant code are disabled.

`-Miomutex`
: For directives, critical sections are generated around Fortran I/O statements.

For pragmas, calls to I/O library functions are system-dependent and are not necessarily guaranteed to be thread-safe. I/O library calls within parallel regions should be protected by critical regions, as shown in the examples in Chapter 17, "*OpenMP Reference Information*", to ensure they function correctly on all systems.

Directive and Pragma Summary Table

The following table provides a brief summary of the directives and pragmas that PGI supports. For complete information on these statements and examples, refer to Chapter 17, "*OpenMP Reference Information*".

Note

In the table, the values in uppercase letters are Fortran directives while the names in lowercase letters are C/C++ pragmas.

Table 5.1. Directive and Pragma Summary Table

Fortran Directive and C/C++ Pragma	Description
ATOMIC and atomic	Semantically equivalent to enclosing a single statement in the CRITCIAL...END CRITICAL directive or critical pragma. Note: Only certain statements are allowed.
BARRIER and barrier	Synchronizes all threads at a specific point in a program so that all threads complete work to that point before any thread continues.
CRITICAL ... END CRITICAL and critical	Defines a subsection of code within a parallel region, a critical section, which is executed one thread at a time.
DO...END DO and for	Provides a mechanism for distribution of loop iterations across the available threads in a parallel region.
C$DOACROSS	Specifies that the compiler should parallelize the loop to which it applies, even though that loop is not contained within a parallel region.
FLUSH and flush	When this appears, all processor-visible data items, or, when a list is present (FLUSH [list]), only those specified in the list, are written to memory, thus ensuring that all the threads in a team have a consistent view of certain objects in memory.
MASTER ... END MASTER and master	Designates code that executes on the master thread and that is skipped by the other threads.
ORDERED and ordered	Defines a code block that is executed by only one thread at a time, and in the order of the loop iterations; this makes the ordered code block sequential, while allowing parallel execution of statements outside the code block.
PARALLEL DO and parallel for	Enables you to specify which loops the compiler should parallelize.
PARALLEL ... END PARALLEL and parallel	Supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.
PARALLEL SECTIONS and parallel sections	Defines a non-iterative work-sharing construct without the need to define an enclosing parallel region.
PARALLEL WORKSHARE ... *END PARALLEL WORKSHARE*	Provides a short form method for including a WORKSHARE directive inside a PARALLEL construct.
SECTIONS ... END SECTIONS and sections	Defines a non-iterative work-sharing construct within a parallel region.
SINGLE ... END SINGLE and single	Designates code that executes on a single thread and that is skipped by the other threads.
TASK and task	Defines an explicit task.

Fortran Directive and C/C++ Pragma	Description
TASKWAIT and taskwait	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
THREADPRIVATE and threadprivate	When a common block or variable that is initialized appears in this directive or pragma, each thread's copy is initialized once prior to its first use.
WORKSHARE ... END WORKSHARE	Provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

Directive and Pragma Clauses

Some directives and pragmas accept clauses that further allow a user to control the scope attributes of variables for the duration of the directive or pragma. Not all clauses are allowed on all directives, so the clauses that are valid are included with the description of the directive and pragma.

The following table provides a brief summary of the clauses associated with OPENMP directives and pragmas that PGI supports.

Table 5.2. Directive and Pragma Clauses Summary Table

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"COLLAPSE (n)"	DO...END DO PARALLEL DO PARALLEL WORKSHARE	parallel for	Specifies how many loops are associated with the loop construct.
"COPYIN (list)"	PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for	Allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region. Then, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"COPYPRIVATE(list)"	SINGLE	single	Specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.
"DEFAULT"	PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for	Specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables.
"FIRSTPRIVATE(list)"	DO PARALLEL PARALLEL DO PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	for parallel parallel for sections single	Specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.
"IF()"	PARALLEL ... END PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for parallel sections	Specifies whether a loop should be executed in parallel or in serial.
"LASTPRIVATE(list)"	DO PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS SECTIONS	parallel parallel for parallel sections sections	Specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration of a for-loop construct or last section of #pragma sections.
"NOWAIT"	DO ... END DO SECTIONS SINGLE WORKSHARE . . . END WORKSHARE	for sections single	Overrides the barrier implicit in a directive.

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"NUM_THREADS"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for parallel sections	Sets the number of threads in a thread team.
"ORDERED"	DO...END DO PARALLEL DO... END PARALLEL DO	parallel for	Required on a parallel FOR statement if an ordered directive is used in the loop.
"PRIVATE"	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	for parallel parallel for parallel sections sections single	Specifies that each thread should have its own instance of a variable.
"REDUCTION"`( {operator | intrinsic } : list)`	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS	for parallel parallel for parallel sections sections	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
"SCHEDULE"`(type [,chunk])`	DO ... END DO PARALLEL DO... END PARALLEL DO	for parallel for	Applies to the FOR directive, allowing the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.
"SHARED"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	parallel parallel for parallel sections	Specifies that one or more variables should be shared among all threads. All threads within a team access the same storage area for shared variables

This clause	Applies to this directive	Applies to this pragma	Has this functionality
"UNTIED"	TASK TASKWAIT	task taskwait	Specifies that any thread in the team can resume the task region after a suspension.

For complete information on these clauses, refer to the OpenMP documentation available on the WorldWide Web.

Run-time Library Routines

User-callable functions are available to the programmer to query and alter the parallel execution environment.

Any C/C++ program unit that invokes these functions should include the statement #include <omp.h>. The `omp.h` include file contains definitions for each of the C/C++ library routines and the required type definitions. For example, to use the `omp_get_num_threads` function, use this syntax:

```
#include <omp.h>
int omp_get_num_threads(void);
```

Note

Unlimited OpenMP thread counts are available in all PGI configurations. The number of threads is unlicensed in the OpenMP run-time libraries - up to the hard limit of 64 threads.

The following table summarizes the run-time library calls.

Note

The Fortran call is shown first followed by the equivalent C/C++ call.

Table 5.3. Run-time Library Routines Summary

Run-time Library Routines with Examples	
omp_get_num_threads Returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the environment variable `OMP_NUM_THREADS` or to the value set by the last previous call to **omp_set_num_threads()**.	
Fortran	`integer function omp_get_num_threads()`
C/C++	`int omp_get_num_threads(void);`

Run-time Library Routines with Examples	
omp_set_num_threads Sets the number of threads to use for the next parallel region. This subroutine or function can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine or function that is called from within a parallel region, the results are undefined. Further, this subroutine or function has precedence over the `OMP_NUM_THREADS` environment variable.	
Fortran	`subroutine omp_set_num_threads(scalar_integer_exp)`
C/C++	`void omp_set_num_threads(int num_threads);`
omp_get_thread_num Returns the thread number within the team. The thread number lies between 0 and **omp_get_num_threads()-1**. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.	
Fortran	`integer function omp_get_thread_num()`
C/C++	`int omp_get_thread_num(void);`
omp_get_ancestor_thread_num Returns, for a given nested level of the current thread, the thread number of the ancestor.	
Fortran	`integer function omp_get_ancestor_thread_num(level)` `integer level`
C/C++	`int omp_get_ancestor_thread_num(int level);`
omp_get_active_level Returns the number of enclosing active parallel regions enclosing the task that contains the call. PGI currently supports only one level of active parallel regions, so the return value currently is 1.	
Fortran	`integer function omp_get_active_level()`
C/C++	`int omp_get_active_level(void);`
omp_get_level Returns the number of parallel regions enclosing the task that contains the call.	
Fortran	`integer function omp_get_level()`
C/C++	`int omp_get_level(void);`
omp_get_max_threads Returns the maximum value that can be returned by calls to **omp_get_num_threads()**. If **omp_set_num_threads()** is used to change the number of processors, subsequent calls to **omp_get_max_threads()** return the new value. Further, this function returns the maximum value whether executing from a parallel or serial region of code.	
Fortran	`integer function omp_get_max_threads()`
C/C++	`void omp_get_max_threads(void);`

Run-time Library Routines with Examples	
omp_get_num_procs	
Returns the number of processors that are available to the program	
Fortran	`integer function omp_get_num_procs()`
C/C++	`int omp_get_num_procs(void);`
omp_get_stack_size	
Returns the value of the OpenMP internal control variable that specifies the size that is used to create a stack for a newly created thread. This value may *not* be the size of the stack of the current thread.	

Fortran	`!omp_get_stack_size interface`
	`function omp_get_stack_size ()`
	`use omp_lib_kinds`
	`integer ( kind=OMP_STACK_SIZE_KIND )`
	`:: omp_get_stack_size`
	`end function omp_get_stack_size`
	`end interface`

C/C++	`size_t omp_get_stack_size(void);`
omp_set_stack_size	
Changes the value of the OpenMP internal control variable that specifies the size to be used to create a stack for a newly created thread. The integer argument specifies the stack size in kilobytes. The size of the stack of the current thread cannot be changed. In the PGI implementation, all OpenMP or auto-parallelization threads are created just prior to the first parallel region; therefore, only calls to `omp_set_stack_size()` that occur prior to the first region have an effect.	
Fortran	`subroutine omp_set_stack_size(integer(KIND=OMP_STACK_SIZE_KIND))`
C/C++	`void omp_set_stack_size(size_t);`
omp_get_team_size	
Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs.	
Fortran	`integer function omp_get_team_size (level)` `integer level`
C/C++	`integer omp_get_team_size(int level);`

Run-time Library Routines with Examples	
omp_in_parallel	
Returns whether or not the call is within a parallel region. Returns `.TRUE.` for directives and non-zero for pragmas if called from within a parallel region and `.FALSE.` for directives and zero for pragmas if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating `.FALSE.` for directives and zero for pragmas, the function returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`logical function omp_in_parallel()`
C/C++	`int omp_in_parallel(void);`
omp_set_dynamic	
Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect.	
Fortran	`subroutine omp_set_dynamic(scalar_logical_exp)`
C/C++	`void omp_set_dynamic(int dynamic_threads);`
omp_get_dynamic	
Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`logical function omp_get_dynamic()`
C/C++	`void omp_get_dynamic(void);`
omp_set_nested	
Allows enabling/disabling of nested parallel regions. This function is recognized, but currently has no effect.	
Fortran	`subroutine omp_set_nested(nested)` `  logical nested`
C/C++	`void omp_set_nested(int nested);`
omp_get_nested	
Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.` for directives and zero for pragmas.	
Fortran	`logical function omp_get_nested()`
C/C++	`int omp_get_nested(void);`

Run-time Library Routines with Examples	
omp_set_schedule	
Set the value of the run_sched_var.	

Fortran	`subroutine omp_set_schedule(kind, modifier)`
	`  include 'omp_lib_kinds.h'`
	`  integer (kind=omp_sched_kind) kind`
	`  integer modifier`

C/C++	`double omp_set_schedule()`
omp_get_schedule	
Retrieve the value of the run_sched_var.	

Fortran	`subroutine omp_get_schedule(kind, modifier)`
	`  include 'omp_lib_kinds.h'`
	`  integer (kind=omp_sched_kind) kind`
	`  integer modifier`

C/C++	`double omp_get_schedule()`
omp_get_wtime	
Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value for directives and as a floating-point double value for pragmas. Times returned are per-thread times, and are not necessarily globally consistent across all threads.	
Fortran	`double precision function omp_get_wtime()`
C/C++	`double omp_get_wtime(void)`
omp_get_wtick	
Returns the resolution of omp_get_wtime(), in seconds, as a DOUBLE PRECISION value for Fortran directives and as a floating-point double value for C/C++ pragmas.	
Fortran	`double precision function omp_get_wtick()`
C/C++	`double omp_get_wtick();`
omp_init_lock	
Initializes a lock associated with the variable lock for use in subsequent calls to lock routines. The initial state of the lock is unlocked. If the variable is already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_init_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`void omp_init_lock(omp_lock_t *lock);` `void omp_init_nest_lock(omp_nest_lock_t *lock);`

Run-time Library Routines with Examples	
omp_destroy_lock Disassociates a lock associated with the variable.	

Fortran	`subroutine omp_destroy_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`void omp_destroy_lock(omp_lock_t *lock);` `void omp_destroy_nest_lock(omp_nest_lock_t *lock);`
omp_set_lock Causes the calling thread to wait until the specified lock is available. The thread gains ownership of the lock when it is available. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_set_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`void omp_set_lock(omp_lock_t *lock);` `void omp_set_nest_lock(omp_nest_lock_t *lock);`
omp_unset_lock Causes the calling thread to release ownership of the lock associated with `integer_var`. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_unset_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`#include <omp.h> void omp_unset_lock(omp_lock_t *lock);` `void omp_unset_nest_lock(omp_nest_lock_t *lock);`
omp_test_lock Causes the calling thread to try to gain ownership of the lock associated with the variable. The function returns `.TRUE.` for directives and non-zero for pragmas if the thread gains ownership of the lock; otherwise it returns `.FALSE.` for directives and zero for pragmas. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`logical function omp_test_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

C/C++	`int omp_test_lock(omp_lock_t *lock);` `int omp_test_nest_lock(omp_nest_lock_t *lock);`

Environment Variables

You can use OpenMP environment variables to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives and pragmas.

The following summary table is a quick reference for the OpenMP environment variables that PGI uses. For detailed descriptions of each of these variables, refer to "OpenMP Environment Variables," on page 308.

Table 5.4. OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
OMP_DYNAMIC	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_NESTED		Currently has no effect. Typically specifies the maximum number of nested parallel regions.
OMP_MAX_ACTIVE_LEVELS	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) nested parallelism.
OMP_NUM_THREADS	1	Specifies the number of threads to use during execution of parallel regions.
OMP_SCHEDULE	STATIC with chunk size of 1	Specifies the type of iteration scheduling and optionally the chunk size to use for *omp for* and *omp parallel for* loops that include the run-time schedule clause. The supported schedule types, which can be specified in upper- or lower-case are static, dynamic, guided, and auto.
OMP_STACKSIZE		Overrides the default stack size for a newly created thread.
OMP_THREAD_LIMIT	64	Specifies the absolute maximum number of threads that can be used in a program.
OMP_WAIT_POLICY	ACTIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

Chapter 6. Using MPI

Message Passing Interface (MPI) is an industry-standard application programming interface designed for rapid data exchange between processors in a cluster application. MPI is computer software used in computer clusters that allows many computers to communicate with one another.

PGI provides MPI support with PGI compilers and tools. You can build MPI applications on Windows using Microsoft's implementation of MPI, MSMPI, on Mac OS X using OpenMPI, and on Linux using MPICH-1, MPICH-2, MVAPICH, OpenMPI, and HP-MPI. PGI Workstation on Linux includes MPICH-1; PGI Workstation on Mac OS X includes OpenMPI; and PGI CDK on Linux includes MPICH-1, MPICH-2, and MVAPICH. This chapter describes how to use these capabilities and some of their limitations.

The debugger and profiler are enabled to support MPI applications running locally with a limited number of processes. The *PGI Tools Guide* describes the MPI-enabled tools in detail:

- PGPROF graphical MPI/OpenMP/multi-thread performance profiler.
- PGDBG graphical MPI/OpenMP/multi-thread symbolic debugger.

MPI Overview

This section contains general information applicable to various MPI implementations. For specific information, refer to the implementation-specific sections later in this chapter.

> **Note**
>
> Due to complexities in the OpenMPI implementation, the `-Mmpi=` openmpi is not supported for OpenMPI. To build using OpenMPI, use the OpenMPI-supplied wrappers mpicc, mpic++, mpif77, or mpif90 to compile and link. Configure these compiler wrappers to use PGI compilers as described in the OpenMPI documentation and the PGI Installation Guide.

MPI is a set of function calls and libraries that are used to send messages between multiple processes. These processes can be located on the same system or on a collection of distributed servers. Unlike OpenMP, the distributed nature of MPI allows it to work in almost any parallel environment. Further, distributed execution of a program does not necessarily mean that you must run your MPI job on many machines.

Compiling and Linking MPI Applications

The PGI compilers provide an option, `-Mmpi=`, to make building MPI applications more convenient by adding the MPI include and library directories to the compiler's include and library search paths. The compiler determines the location of these directories using various mechanisms described in the MPI implementation-specific sections later in this chapter.

Table 6.1 lists the `-Mmpi=` suboptions for each of the supported implementations.

Table 6.1. MPI Implementation Options

This MPI implementation...	Requires this option...
MPICH-1	`-Mmpi=mpich1`
MPICH-2	`-Mmpi=mpich2`
MVAPICH	`-Mmpi=mvapich1`
HP-MPI	`-Mmpi=hpmpi`
MSMPI	`-Mmpi=msmpi`
OpenMPI	`-Mmpi` not supported. Use compiler wrappers.

Note

When you build an MPI application, you must use these options in both the compile and link steps.

Debugging MPI Applications

The PGI debugger, PGDBG, provides support for symbolic debugging of MPI applications. In PGI Workstation, this support is limited in two ways:

- The application processes must ron on a system where PGI Workstation is installed.
- The number of processes is limited, controlled by the license key.

For all implementations of MPI except MPICH-1, you use the PGDBG command to initiate an MPI debugging session. For example, you might invoke an HP-MPI session as follows:

```
% pgdbg -mpi:/opt/hpmpi/bin/mpirun -np 4 my_mpi_app
```

For specific information on how to initiate a debugging session for your instance of MPI, refer to the implementation-specific sections available later in this chapter.

PGDBG can display the contents of message queues for instances of MPI that have been configured to support that feature. The version of MPICH-1 provided with PGI Workstation as well as HP-MPI function properly. If you are using MPICH-2 or MVAPICH, these must be built and configured correctly. Further, MSMPI does not support dumping of message queue contents.

For more information on MPI and displaying message queues, refer to the documentation for your specific implementation.

Profiling MPI Applications

The PGI performance profiler, PGPROF, provides support for profiling MPI applications. The number of processes that can be profiled is limited by your license. PGPROF instrumentation is inserted into the program by the compiler, and after the program is executed, the PGPROF profiler can display MPI message count statistics as they relate to the source code of the application and the time spent in those portions of the application.

To create and view a performance profile of your MPI application, you must first build an instrumented version of the application using the `-Mprof=` option to specify one of the MPI implementations. The `-Mprof=` option requires that you use another profiling sub-option in conjunction with the MPI implementation suboptions, listed in Table 6.2.

Note

When you build an MPI application for profiling, you must use these options in both the compile and link steps.

Table 6.2. MPI Profiling Options

This MPI implementation...	Requires this profiling option...
MPICH-1	`-Mprof=mpich1,{func|hwcts|lines|time}`
MPICH-2	`-Mprof=mpich2,{func|hwcts|lines|time}`
MVAPICH	`-Mprof=mvapich1,{func|hwcts|lines|time}`
HP-MPI	`-Mprof=hpmpi,{func|hwcts|lines|time}`
MSMPI	`-Mprof=msmpi,{func|lines}`
OpenMPI	Use OpenMPI compiler wrappers with `-Mprof=hpmpi,{func|hwcts|lines|time}`

For example, you can use the following command to compile for profiling with MPICH-2:

```
% pgf95 -fast -Mprof=mpich2,func my_mpi_app.f90
```

Note

The default versions of the compilation scripts, such as `mpicc` and `mpif90`, that are provided by some MPI distributions, do not correctly support the `-Mprof` option. For best results, use the PGI compiler drivers in place of these scripts.

For OpenMPI, refer to the PGI Workstation Installation Guide for more information on configuring the OpenMPI compiler wrappers to work with the PGI compilers on your system.

Once you have built an instrumented version of your MPI application, running it produces the profile data. For specific details on using PGPROF to view the profile data, refer to the *PGI Tools Guide*.

Using MPICH-1 on Linux

PGI Workstation and CDK for Linux includes MPICH-1 libraries, tools, and licenses required to compile, execute, profile, and debug MPI programs. PGI Workstation can be installed on a single system, and that

system can be treated as if it is a small cluster. The MPI profiler and debugger are limited to processes on a single system in PGI Workstation. The PGI CDK supports general development on clusters.

Example

Example 6.1. MPI Hello World Example

The following MPI "hello world" example program uses MPICH-1.

```
% cd my_example_dir
% cp -r $PGI/linux86/9.0-0/EXAMPLES/MPI/mpihello .
% cd mpihello
% pgf77 -o mpihello mpihello.f -Mmpi=mpich1
```

```
% mpirun mpihello
Hello world! I'm node 0
```

```
% mpirun -np 4 mpihello
Hello world! I'm node 0
Hello world! I'm node 2
Hello world! I'm node 1
Hello world! I'm node 3
```

If you want to build your MPI application using the instance of MPICH-1 installed with the PGI compilers, just use the `-Mmpi=mpich1` option, or the `-Mprof=mpich1` option to instrument for MPICH-1 profiling.

To use a different instance of MPICH-1, set the `MPIDIR` environment variable before invoking the compiler. `MPIDIR` specifies the location of the instance of MPI to use. For example, set `MPIDIR` to the root of the MPICH-1 installation directory that you want to use, that is, the directory that contains `bin`, `include`, `lib`, and so on.

Using MPICH-2 on Linux

PGI CDK for Linux includes MPICH-2 libraries, tools, and licenses required to compile, execute, profile, and debug MPI programs.

If you want to build your MPI application using the instance of MPICH-2 installed with the PGI compilers, just add the `-Mmpi=mpich2` option to the compilation and link steps, or you can use the `-Mprof=mpich2` option to instrument for MPICH-2 profiling. The `-Mmpi=mpich2` option automatically sets up the include and library paths to use the MPICH-2 headers and libraries. For example, you can use the following command to compile for profiling with MPICH-2:

```
% pgf95 -fast -Mprof=mpich2,lines my_mpi_app.f90
```

To use a different instance of MPICH-2, set the `MPIDIR` environment variable before invoking and linking with `-Mmpi=mpich2`. `MPIDIR` specifies the location of the instance of MPI to use. For example, set `MPIDIR` to the root of the MPICH-2 installation directory that you want to use, that is, the directory that contains `bin`, `include`, `lib`, and so on.

Using MVAPICH on Linux

PGI CDK for Linux includes MVAPICH libraries, tools, and licenses required to compile, execute, profile, and debug MPI programs.

If you want to build your MPI application using the instance of MVAPICH installed with the PGI compilers, just add the `-Mmpi=mvapich1` option to the compilation and link steps, or you can use the `-Mprof=mvapich1` option to instrument for MVAPICH profiling. The `-Mmpi=mvapich1` option automatically sets up the include and library paths to use the MVAPICH headers and libraries. For example, you can use the following command to compile for profiling with MVAPICH:

```
% pgf95 -fast -Mprof=mvapich1,func my_mpi_app.f90
```

To use a different instance of MVAPICH, set the `MPIDIR` environment variable before invoking and linking with `-Mmpi=mvapich1`. `MPIDIR` specifies the location of the instance of MPI to use. For example, set `MPIDIR` to the root of the MVAPICH installation directory that you want to use, that is, the directory that contains `bin`, `include`, `lib`, and so on.

Using HP-MPI on Linux

If you have an instance of HP-MPI available on the Linux system on which you have the PGI compilers installed, you can compile, run, debug, and profile locally on your system using that instance of HP-MPI.

HP-MPI Installation Directory

By default, HP-MPI on Linux is installed in `/opt/hpmpi`.

- If your instance of HP-MPI is installed in `/opt/hpmpi`, the default location, then compiling and linking with the options `-Mmpi=hpmpi` and `-Mprof=hpmpi` automatically brings in the appropriate include files and libraries.
- If your instance of HP-MPI is installed in a directory other than the default location, then you must set the HP-MPI environment variable `MPI_ROOT` to the alternate directory where HP-MPI is installed. `MPI_ROOT` specifies the location of the instance of HP-MPI to use when compiling your application.

Multi-threaded Application Support

The `-Mmpi=hpmpi` and `-Mprof=hpmpi` options use the *non-thread-compliant* version of HP-MPI. This library is sufficient for applications that do not make MPI calls simultaneously from multiple threads.

The *thread-compliant* version is not supported by these options, although you can link to the library directly using the `-L` and `-l` options.

For more information on using HP-MPI in a multi-threaded application, refer to the HP-MPI documentation.

Using OpenMPI on Linux

PGI does not include a version of OpenMPI on Linux that is preconfigured. However, you can configure your system for OpenMPI by following the instructions in the PGI Workstation Installation Guide.

Compiling using OpenMPI

To build the application, use the OpenMPI compiler wrappers: mpicc, mpic++, mpif77, and mpif90. These wrappers automatically set up the compiler commands with the correct include file search paths, library

directories, and link libraries. Unlike other MPI distributions, the PGI compiler drivers do not directly support the `-Mmpi=openmpi` option for OpenMPI.

Generate MPI Profile Data

To build an application that generates MPI profile data suitable for use with PGPROF, use the OpenMPI compiler wrappers with the `-Mprof=func`, `-Mprof=lines`, or `-Mprof=time` option. On linux86-64 configured with PAPI, you may also use `-Mprof=hwcts`. For more information on profiling with PAPI, refer to the PGI Tools Guide.

Unlike for other MPI distributions, the PGI compiler drivers do not directly support the `-Mprof=openmpi` option.

Using MSMPI on Windows

If you have an implementation of MSMPI available on the Windows system on which you have PGI Workstation installed, you can compile, run, debug, and profile locally on your system using that instance of MSMPI. In general these instructions apply to a system where the Microsoft HPC Pack 2008 SDK is installed as a development tool, as opposed to an actual HPC Server system.

MSMPI Environment

When the Microsoft HPC Pack 2008 SDK is installed, some system environment variables are set. Further, there are two environment variables available to help you specify directory locations associated with using MSMPI on Windows: `CCP_HOME` and `CCP_SDK`.

- `CCP_HOME` specifies the root directory of the Microsoft cluster management software for systems on which the Microsoft HPC Pack 2008 is installed.
- `CCP_SDK` specifies the root directory of the MSMPI software for systems on which Microsoft's HPC Pack 2008 SDK is installed.

If the appropriate environment variable is set for the version of MSMPI that you are using, then both the options `-Mmpi=msmpi` and `-Mprof=msmpi` automatically bring in the appropriate include files and libraries.

Compiling using MSMPI

To compile the application, use the `-Mmpi=msmpi` option. This option inserts options into the compile and link lines to pick up the MSMPI headers and libraries.

Generate MPI Profile Data

To build an application that generates MPI profile data, use the `-Mprof=msmpi` option. This option performs MPICH-style profiling for Microsoft MPI.

The profile data generated by running an application built with the option `-Mprof=msmpi` contains information about the number of sends and receives, as well as the number of bytes sent and received, correlated with the source location associated with the sends and receives. You must use `-Mprof=msmpi` in

conjunction with `-Mprof=func` or `-Mprof=lines`. When invoked using this type of profile data, PGPROF automatically displays MPI statistics.

Using OpenMPI on Mac OS X

PGI Workstation for Mac OS X includes a version of OpenMPI preconfigured for use with the PGI compilers.

Compiling using OpenMPI

To build the application, use the OpenMPI compiler wrappers: mpicc, mpic++, mpif77, and mpif90. These wrappers automatically set up the compiler commands with the correct include file search paths, library directories, and link libraries. Unlike other MPI distributions, the PGI compiler drivers do not directly support the `-Mmpi=openmpi` option for OpenMPI.

Generate MPI Profile Data

Yo build an application that generates MPI profile data suitable for use with PGPROF, use the OpenMPI compiler wrappers with the `-Mprof=func` or `-Mprof=lines` option.

Note

`-Mprof=time` and `-Mprof=hwcts` are not supported on Mac OS X.

Unlike for other MPI distributions, the PGI compiler drivers do not directly support the `-Mprof=openmpi` option.

Chapter 7. Using an Accelerator

An accelerator is a special-purpose co-processor attached to a CPU and to which the CPU can offload data and executable kernels to perform compute-intensive calculations. This chapter describes a collection of compiler directives used to specify regions of code in Fortran and C programs that can be offloaded from a *host* CPU to an attached *accelerator*.

Overview

The programming model and directives described in this chapter allow programmers to create high-level *host+accelerator* programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. Rather, all of these details are implicit in the programming model and are managed by the PGI Fortran and C accelerator compilers.

The method described provides a model for accelerator programming that is portable across operating systems and various host CPUs and accelerators. The directives allow a programmer to migrate applications incrementally to accelerator targets using standards-compliant Fortran or C.

This programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator region, guidance on mapping of loops onto an accelerator, and similar performance-related details.

Components

The PGI Accelerator compiler technology includes the following components:

- PGF95 auto-parallelizing accelerator-enabled Fortran 90/95 compiler.
- PGCC auto-parallelizing accelerator-enabled ANSI C99 and K&R C compiler.
- A simple command-line tool to detect whether the system has an appropriate GPU or accelerator card.

No accelerator-enabled debugger or profiler is included with this release.

Availability

The PGI Fortran & C Accelerator compilers are available only on x64 processor-based workstations and servers running 64-bit Linux operating systems, with an attached NVIDIA CUDA-enabled GPU or Tesla card.

These compilers target only a limited number and type of x64+GPU platforms. All examples included in this chapter are developed and presented on such a platform. For a list of supported GPUs, refer to the Accelerator Installation and Supported Platforms list in the latest PGI Release Notes.

User-directed Accelerator Programming

In user-directed accelerator programming the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The bulk of a user's program, as well as regions containing constructs that are not supported on the targeted accelerator, are executed on the host. This chapter concentrates on specification of loops and regions of code to be offloaded to an accelerator.

Features Not Covered or Implemented

This chapter does not describe features or limitations of the host programming environment as a whole. Further, it does not cover automatic detection and offloading of regions of code to an accelerator by a compiler or other tool, nor does it cover targeting of accelerator regions to multiple accelerators attached to a single host. While future versions of the PGI compilers may allow for automatic offloading, multiple accelerators of the same type, or multiple accelerators of different types, these features are not currently supported.

Terminology

Clear and consistent terminology is important in describing any programming model. This section provides definitions of the terms required for you to effectively use this chapter and the associated programming model.

Accelerator
: a special-purpose co-processor attached to a CPU and to which the CPU can offload data and executable kernels to perform compute-intensive calculations.

Compute intensity
: for a given loop, region, or program unit, the ratio of the number of arithmetic operations performed on computed data divided by the number of memory transfers required to move that data between two levels of a memory hierarchy.

CUDA
: stands for Compute Unified Device Architecture; the CUDA environment from NVIDIA is a C-like programming environment used to explicitly control and program an NVIDIA GPU.

Device
: a general reference to any type of accelerator.

Device memory
: memory attached to an accelerator which is physically separate from the host memory.

Directive
: in C, a #pragma, or in Fortran, a specially formatted comment statement, that is interpreted by a compiler to augment information about or specify the behavior of the program.

GPU
: a Graphics Processing Unit; one type of accelerator device.

GPGPU
: General Purpose computation on Graphics Processing Units.

Host
: the main CPU that in this context has an attached accelerator device. The host CPU controls the program regions and data loaded into and executed on the device.

Loop trip count
: the number of times a particular loop executes.

OpenCL - Open Compute Language
: a proposed standard C-like programming environment similar to CUDA that enables portable low-level general-purpose programming on GPUs and other accelerators.

Private data
: with respect to an iterative loop, data which is used only during a particular loop iteration. With respect to a more general region of code, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region.

Structured block
: in C, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

Vector operation
: a single operation or sequence of operations applied uniformly to each element of an array.

System Requirements

To use the PGI Accelerator compiler features, you must install the NVIDIA CUDA components for 64-bit Linux:

- NVIDIA Driver
- CUDA Toolkit
- CUDA SDK

> **Note**
>
> You may download these components from the NVIDIA website at
> `www.nvidia.com/cuda`
>
> These are not PGI products, and are licensed and supported by NVIDIA.

Further, you must be using a 64-bit Linux operating system that is supported by both the current PGI release and by the CUDA software and drivers.

Supported Processors and GPUs

This PGI Accelerator compiler release supports all AMD64 and Intel 64 host processors supported by Release 9.0 or higher of the PGI compilers and tools. Further, you can use the `-tp <target>` flag as documented in that release.

You can also use the `-ta=nvidia` flag to enable the accelerator directives and target the NVIDIA GPU. You can then use the generated code on any system with CUDA installed that has a CUDA-enabled GeForce, Quadro, or Tesla card.

For more information on these flags as they relate to accelerator technology, refer to "Applicable Command Line Options," on page 94.

For a complete list of supported GPUs, refer to the NVIDIA website at:

```
www.nvidia.com/object/cuda_learn_products.html
```

You can detect whether the system has CUDA properly installed and has an attached graphics card by running the **pgaccelinfo** command, which is delivered as part of the PGI Accelerator compilers software package.

Installation and Licensing

Note

The PGI Accelerator compilers require a separate license key in addition to a normal PGI Workstation, Server, or CDK license.

Required Files

After installing the PGI Release 9.0-1 or higher compilers and tools, you must create a `sitenvrc` file to tell the compilers the location of the installation directory that contains your NVIDIA CUDA software.

The `sitenvrc` file should contain three lines similar to the following sample. In this example the CUDA software was installed in the `/opt/cuda` directory.

```
set NVOPEN64DIR=/opt/cuda/open64/lib;
set CUDADIR=/opt/cuda/bin;
set CUDALIB=/opt/cuda/lib;
```

The default NVIDIA Compute Capability for generated code is 1.3. You can change the default to 1.0 by adding the following line to the `sitenvrc` file. Adding this line disables generation of NVIDIA GPU code for double precision operations.

```
set COMPUTECAP=10;
```

You must place the `sitenvrc` file in the following directory, where $PGI is the PGI installation directory, which is typically `/opt/pgi` or `/usr/pgi`.

```
$PGI/linux86-64/9.0-1/bin/
```

Command Line Flag

After creating the `sitenvrc` file and acquiring the PGI Accelerator compilers license key, you can use the option `-ta=nvidia` with the `pgf95` or `pgcc` commands.

For more information on the `-ta` flag and the suboptions that relate to the target accelerators, refer to "Applicable Command Line Options," on page 94.

The compiler automatically invokes the necessary CUDA software tools to create the kernel code and embeds the kernels in the Linux object file.

Note

To access the accelerator libraries, you must link an accelerator program with the `-ta` flag as well.

Execution Model

The execution model targeted by the PGI Accelerator compilers is host-directed execution with an attached accelerator device, such as a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The accelerator device executes kernels, which may be as simple as a tightly-nested loop, or as complex as a subroutine, depending on the accelerator hardware.

Host Functions

Even in accelerator-targeted regions, the host must orchestrate the execution; it

- allocates memory on the accelerator device
- initiates data transfer
- sends the kernel code to the accelerator
- passes kernel arguments
- queues the kernel
- waits for completion
- transfers results back to the host
- deallocates memory

Note

In most cases, the host can queue a sequence of kernels to be executed on the device, one after the other.

Levels of Parallelism

Most current GPUs support two levels of parallelism:

- an outer *doall* (fully parallel) loop level
- an inner *synchronous* (SIMD or vector) loop level

Each level can be multidimensional with 2 or 3 dimensions, but the domain must be strictly rectangular. The *synchronous* level may not be fully implemented with SIMD or vector operations, so explicit synchronization is supported and required across this level. No synchronization is supported between parallel threads across the *doall* level.

The execution model on the device side exposes these two levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization across iterations. All fully parallel loops can be scheduled for either doall or synchronous parallel execution, but by definition SIMD vector loops that require synchronization can only be scheduled for synchronous parallel execution.

Memory Model

The most significant difference between a *host-only* program and a *host+accelerator* program is that the memory on the accelerator can be completely separate from host memory, which is the case on most current GPUs. For example:

- The host cannot read or write accelerator memory by reference because it is not mapped into the virtual memory space of the host.
- All data movement between host memory and accelerator memory must be performed by the host through runtime library calls that explicitly move data between the separate memories.
- It is not valid to assume the accelerator can read or write host memory, though this may be supported by accelerators in the future.

Separate Host and Accelerator Memory Considerations

The concept of separate host and accelerator memories is very apparent in low-level accelerator programming models such as CUDA or OpenCL, in which data movement between the memories dominates user code. In the PGI Accelerator programming model, data movement between the memories is implicit and managed by the compiler.

The programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and accelerator memory determines the compute intensity required to effectively accelerate a given region of code.
- Limited size of accelerator memory may prohibit offloading of regions of code that operate on very large amounts of data.

Accelerator Memory

On the accelerator side, current GPUs implement a weak memory model. In particular, they do not support memory coherence between threads unless those threads are parallel only at the synchronous level and the memory operations are separated by an explicit barrier. Otherwise, if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, the hardware does not guarantee the results. While the results of running such a program might be inconsistent, it is not accurate to say that the results are incorrect. By definition, such programs are defined as being in error. While a compiler can detect some potential errors of this nature, it is nonetheless possible to write an accelerator region that produces inconsistent numerical results.

Cache Management

Some current GPUs have a software-managed cache, some have hardware-managed caches, and most have hardware caches that can be used only in certain situations and are limited to read-only data. In low-level programming models such as CUDA or OpenCL, it is up to the programmer to manage these caches. However, in the PGI Accelerator programming model, the compiler manages these caches using hints from the programmer in the form of directives.

Running an Accelerator Program

Running a program that has accelerator directives and was compiled and linked with the `-ta=nvidia` flag is the same as running the program compiled without the `-ta=nvidia` flag. The program looks for and dynamically loads the CUDA libraries. If the libraries are not available, or if they are in a different directory than they were when the program was compiled, you may need to append the CUDA library directory to your LD_LIBRARY_PATH environment variable.

When your program reaches its first accelerator region, there is a 0.5 to 1.5 second pause, which is the cost to acquire a handle to the GPU and allocate static resources. After that, there is no overhead to execute accelerator regions.

If you run an accelerated program on a system without a CUDA-enabled NVIDIA GPU, or without the CUDA software installed in a directory where the runtime library can find it, the program fails at runtime with an error message.

If you set the environment variable ACC_NOTIFY to a nonzero integer value, the runtime library prints a line to standard error every time it launches a kernel.

Directives

This section provides an overview of the Fortran and C directives used to delineate accelerator regions and to augment information available to the compiler for scheduling of loops and classification of data. For complete descriptions of each accelerator directive, refer to "PGI Accelerator Directives," on page 311.

Enable Accelerator Directives

PGI Accelerator compilers enable accelerator directives with the `-ta` command line option. For more information on this option as it relates to the Accelerator, refer to "Applicable Command Line Options," on page 94.

Note

The syntax used to define directives allows compilers to ignore accelerator directives if support is disabled or not provided.

_ACCEL macro

The _ACCEL macro name is defined to have a value yyyymm where yyyy is the year and mm is the month designation of the version of the Accelerator directives supported by the implementation. For example, the version for May, 2009 is 200905. This macro must be defined by a compiler when accelerator directives are enabled.

Format

The specific format of the directive depends on the language and the format or form of the source.

Directives include a name and clauses, and the format of the directive depends on the type:

- C directives, described in "C Directives"

- Free-format Fortran directives, described in "Free-Form Fortran Directives"
- Fixed-format Fortran directives, described in "Fixed-Form Fortran Directives"

Note

This document uses free form for all PGI Accelerator compiler Fortran directive examples.

Rules

The following rules apply to all PGI Accelerator compiler directives:

- Only one directive-name can be specified per directive.
- The order in which clauses appear is not significant.
- Clauses may be repeated unless otherwise specified.
- For clauses that have a list argument; a list is a comma-separated list of variable names, array names, or, in some cases, subarrays with subscript ranges.

C Directives

In C, PGI Accelerator compiler directives are specified using `#pragma`

Syntax

The syntax of a PGI Accelerator compiler directive is:

```
#pragma acc directive-name [clause [,clause]…] new-line
```

Rules

In addition to the general directive rules, the following rules apply to PGI Accelerator compiler C directives:

- Each directive starts with `#pragma acc`.
- The remainder of the directive follows the C conventions for pragmas.
- White space may be used before and after the #; white space may be required to separate words in a directive.
- Preprocessing tokens following the `#pragma acc` are subject to macro replacement.
- C directives are case sensitive.
- An Accelerator directive applies to the immediately following structured block or loop.

Free-Form Fortran Directives

PGI Accelerator compiler Fortran directives can be either Free-Form or Fixed format directives. Free-Form Accelerator directives are specified with the `!$acc` mechanism.

Syntax

The syntax of directives in free-form source files is:

```
!$acc directive-name [clause [,clause]…]
```

Rules

In addition to the general directive rules, the following rules apply to PGI Accelerator compiler Free-Form Fortran directives:

- The comment prefix (!) may appear in any column, but may only be preceded by white space (spaces and tabs).
- The sentinel (!$acc) must appear as a single word, with no intervening white space.
- Line length, white space, and continuation rules apply to the directive line.
- Initial directive lines must have a space after the sentinel.
- Continued directive lines must have an ampersand (&) as the last nonblank character on the line, prior to any comment placed in the directive.
- Comments may appear on the same line as the directive, starting with an exclamation point and extending to the end of the line.
- If the first nonblank character after the sentinel is an exclamation point, the line is ignored.
- Directives are case-insensitive.
- Directives cannot be embedded within continued statements.
- Statements must not be embedded within continued directives.

Fixed-Form Fortran Directives

PGI Accelerator Fortran directives can be either Free-Form or Fixed format directives.

Syntax

Fixed-Form Accelerator directives are specified using one of three formats.

```
!$acc directive-name [clause [,clause]…]
c$acc directive-name [clause [,clause]…]
*$acc directive-name [clause [,clause]…]
```

Rules

In addition to the general directive rules, the following rules apply to Accelerator Fixed-Form Fortran directives:

- The sentinel (!acc, cacc, or *$acc) must occupy columns 1-5.
- Fixed form line length, white space, continuation, and column rules apply to the directive line.

- Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or zero in column 6.
- Comments may appear on the same line as a directive, starting with an exclamation point on or after column 7 and continuing to the end of the line.
- Directives are case-insensitive.
- Directives cannot be embedded within continued statements.
- Statements must not be embedded within continued directives.

Accelerator Directive Summary

Table 7.1 lists and briefly describes each of the accelerator directives that PGI currently supports. For a complete description of each directive, refer to "PGI Accelerator Directives," on page 311.

Table 7.1. PGI Accelerator Directive Summary Table

This directive...	Accepts these clauses...	Has this functionality...
Accelerator Region Directive	if copy (list) copyin(list) copyout(list) local(list)	Defines the region of the program that should be compiled for execution on the accelerator device.

C Syntax
`#pragma acc region [clause [, clause]…] new-line`
`structured block`
Fortran Syntax
`!$acc region [clause [, clause]…]`
`structured block`
`!$acc end region`		

Accelerator Loop Mapping Directive	host [(width)] kernel parallel [(width)] private(list) seq [(width)] vector [(width)]	Describes what type of parallelism to use to execute the loop and declare loop-private variables and arrays. Applies to a loop which must appear on the following line.

C Syntax
`#pragma acc for [clause [,clause]...]new-line`
`for loop`
Fortran Syntax
`!$acc do [clause [,clause]…]`
`do loop`		

This directive...	Accepts these clauses...	Has this functionality...
Combined Directives	Any clause that is allowed on a region directive or a loop directive is allowed on a combined directive.	Is a shortcut for specifying a loop directive nested immediately inside an accelerator region directive. The meaning is identical to explicitly specifying a region construct containing a loop directive.

C Syntax

```
#pragma acc region for [clause [, clause]...] new-line
              for loop
```

Fortran Syntax

```
   !$acc region do [clause [, clause]...]
              do loop
```

Accelerator Directive Clauses

This section contains two tables, "Region Clauses Summary" and "Region Clauses Summary" that provide alphabetical listings and brief descriptions of each clause that is applicable for Accelerator directives:

For more information on the restrictions and use of each clause, refer to "PGI Accelerator Directive Clauses," on page 314.

Table 7.2 describes the clauses associated with the accelerator region directive.

Table 7.2. Region Clauses Summary

Use this clause...	to do this...
copy clause	Declares that the variables, arrays or subarrays in the list have values in the host memory that need to be copied to the accelerator memory, and are assigned values on the accelerator that need to be copied back to the host.
copyin clause	Declares that the variables, arrays or subarrays in the list have values in the host memory that need to be copied to the accelerator memory.
copyout clause	Declares that the variables, arrays, or subarrays in the list are assigned or contain values in the accelerator memory that need to be copied back to the host memory at the end of the accelerator region.
if clause	When present, tells the compiler to generate two copies of the region - one for the accelerator, one for the host - and to generate code to decide which copy to execute.
local clause	Declares that the variables, arrays or subarrays in the list need to be allocated in the accelerator memory, but the values in the host memory are not needed on the accelerator, and the values computed and assigned on the accelerator are not needed on the host.

Table 7.3 describes the clauses associated with the accelerator loop mapping directive.

Table 7.3. Loop Clauses Summary

Use this clause...	to do this...
host clause	Tells the compiler to execute the loop sequentially on the host processor.
kernel clause	Tells the compiler that the body of this loop is to be the body of the computational kernel. Any loops contained within the kernel loop are executed sequentially on the accelerator.
parallel clause	Tells the compiler to execute this loop in parallel mode on the accelerator. There may be a target-specific limit on the number of iterations in a parallel loop or on the number of parallel loops allowed in a given kernel
private clause	Declares that the variables, arrays, or subarrays in the list argument need to be allocated in the accelerator memory with one copy for each iteration of the loop.
seq clause	Tells the compiler to execute this loop sequentially on the accelerator. There is no maximum number of iterations for a seq schedule.
vector clause	Tells the compiler to execute this loop in vector mode on the accelerator. There may be a target-specific limit on the number of iterations in a vector loop, the aggregate number of iterations in all vector loops, or the number of vector loops allowed in a kernel.

PGI Accelerator Compilers Runtime Libraries

This section provides an overview of the user-callable functions and library routines that are available for use by programmers to query the accelerator features and to control behavior of accelerator-enabled programs at runtime.

Note

In Fortran, none of the PGI Accelerator compilers runtime library routines may be called from a PURE or ELEMENTAL procedure.

Runtime Library Definitions

There are separate runtime library files for C and for Fortran.

C Runtime Library Files

In C, prototypes for the runtime library routines are available in a header file named `accel.h`. All the library routines are extern functions with "C" linkage. This file defines:

- The prototypes of all routines in this section.
- Any data types used in those prototypes, including an enumeration type to describe types of accelerators.

Fortran Runtime Library Files

In Fortran, interface declarations are provided in a Fortran include file named `accel_lib.h` and in a Fortran module named `accel_lib`. These files define:

- Interfaces for all routines in this section chapter.
- Integer parameters to define integer kinds for arguments to those routines.
- Integer parameters to describe types of accelerators.
- The integer parameter accel_version with a value `yyyymm` where `yyyy` and `mm` are the year and month designations of the version of the Accelerator programming model supported. This value matches the value of the preprocessor variable `_ACCEL`.

Runtime Library Routines

Table 7.4 lists and briefly describes the supported PGI Accelerator compilers runtime library routines. For a complete description of these routines, refer to "PGI Accelerator Runtime Routines," on page 319.

Table 7.4. Accelerator Runtime Library Routines

This Runtime Library Routine...	Does this...
acc_get_device	Returns the type of accelerator device being used.
acc_get_num_devices	Returns the number of accelerator devices of the given type attached to the host.
acc_init	Connects to and initializes the accelerator device and allocates control structures in the accelerator library.
acc_set_device_num	Tells the runtime which device to use among those attached of the given type.

Environment Variables

PGI supports environment variables that modify the behavior of accelerator regions. This section defines the user-setable environment variables used to control behavior of accelerator-enabled programs at execution. These environment variables must comply with these rules:

- The names of the environment variables must be upper case.
- The values assigned environment variables are case insensitive and may have leading and trailing white space.
- The behavior is implementation-defined if the values of the environment variables change after the program has started, even if the program itself modifies the values.

Table 7.5 lists and briefly describes the Accelerator environment variables that PGI supports.

Table 7.5. Accelerator Environment Variables Routines

This environment variable...	Does this...
ACC_DEVICE	Controls which accelerator device to use when executing PGI Unified Binary for accelerators. The value of this environment variable may be the string NVIDIA or HOST.

This environment variable...	Does this...
ACC_DEVICE_NUM	Controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host.
ACC_NOTIFY	When set to a non-negative integer, indicates to print a message for each kernel launched on the device.

Applicable Command Line Options

The following command line options are applicable specifically when working with accelerators.

`-ta`
Use this option to enable recognition of the **!$ACC** directives in Fortran, and **#pragma acc** directives in C.

`-tp`
Use this option to specify the target host processor architecture.

`-Minfo` or `-Minfo=accel`
Use this option to see messages about the success or failure of the compiler in translating the accelerator region into GPU kernels.

The `-ta` flag has the following accelerator-related suboptions:

nvidia
Select NVIDIA accelerator target.

analysis
Perform loop analysis only; do not generate GPU code.

cc10
Generate code for compute capability 1.0.

cc11
Generate code for compute capability 1.1.

cc13
Generate code for compute capability 1.3.

nofma
Do not generate fused multiply-add instructions.

host
Select NO accelerator target. Generate PGI Unified Binary Code, as described "PGI Unified Binary for Accelerators," on page 95.

time
Link in a limited-profiling library, as described in "Profiling Accelerator Kernels," on page 96.

The compiler automatically invokes the necessary CUDA software tools to create the kernel code and embeds the kernels in the Linux object file.

Note

To access accelerator libraries, you must *link* an accelerator program with the `-ta` flag.

PGI Unified Binary for Accelerators

PGI compilers support the PGI Unified Binary feature to generate executables with functions optimized for different host processors, all packed into a single binary. This release extends the PGI Unified Binary technology for accelerators. Specifically, you can generate a single binary that includes two versions of functions:

- one version is optimized for the accelerator
- other version runs on the host processor when the accelerator is not available or when you want to compare host to accelerator execution.

To enable this feature, use the extended `-ta` flag:

```
-ta=nvidia,host
```

This flag tells the compiler to generate two versions of functions that have valid accelerator regions.

- A compiled version that targets the accelerator.
- A compiled version that ignores the accelerator directives and targets the host processor.

If you use the `-Minfo` flag, you get messages similar to the following:

```
s1:
     12, PGI Unified Binary version for -tp=barcelona-64 -ta=host
     18, Generated an alternate loop for the inner loop
         Generated vector sse code for inner loop
         Generated 1 prefetch instructions for this loop
s1:
     12, PGI Unified Binary version for -tp=barcelona-64 -ta=nvidia
     15, Generating copy(b(:,2:90))
         Generating copyin(a(:,2:90))
     16, Loop is parallelizable
     18, Loop is parallelizable
         Parallelization requires privatization of array t(2:90)
         Accelerator kernel generated
         16, !$acc do parallel
         18, !$acc do parallel, vector(256)
             Using register for t
```

The PGI Unified Binary message shows that two versions of the subprogram s1 were generated:

- one for no accelerator (`-ta=host`)
- one for the NVIDIA GPU (`-ta=nvidia`)

At run time, the program tries to load the NVIDIA CUDA dynamic libraries and test for the presence of a GPU. If the libraries are not available or no GPU is found, the program runs the host version.

You can also set an environment variable to tell the program to run on the NVIDIA GPU. To do this, set ACC_DEVICE to the value `NVIDIA` or `nvidia`. Any other value of the environment variable causes the program to use the host version.

Note

The only supported `-ta` targets for this release are `nvidia` and `host`.

Multiple Processor Targets

You can use the `-tp` flag with multiple processor targets along with the `-ta` flag. You will see the following behavior:

- If you specify one `-tp` value and one `-ta` value:

  You see one version of each subprogram generated for that specific target processor and target accelerator.

- If you specify one `-tp` value and multiple `-ta` values:

  The compiler generates two versions of subprograms that contain accelerator regions for the specified target processor and each target accelerator.

- If you specify multiple `-tp` values and one `-ta` value:

  If 2 or more `-tp` values are given, the compiler generates up to that many versions of each subprogram, for each target processor, and each version also targets the selected accelerator.

- If you specify multiple `-tp` values and multiple `-ta` values:

  With 'N' `-tp` values and two `-ta` values, the compiler generates up to N+1 versions of the subprogram. It first generates up to N versions, for each `-tp` value, ignoring the accelerator regions, equivalent to `-ta=host`. It then generates one additional version with the accelerator target.

Profiling Accelerator Kernels

This release supports the command line option:

```
-ta=nvidia,time
```

The *time* suboption links in a timer library, which collects and prints out simple timing information about the accelerator regions and generated kernels.

Example 7.1. Accelerator Kernel Timing Data

```
/proj/qa/tests/accel/bb04.f90
  s1
   15: region entered 1 times
  time(us): total=1490738
                 init=1489138 region=1600
                 kernels=155 data=1445
  w/o init: total=1600 max=1600
                 min=1600 avg=1600
   18: kernel launched 1 times
  time(us): total=155 max=155 min=155 avg=155
```

In this example, a number of things are occurring:

- For each accelerator region, the file name `/proj/qa/tests/accel/bb04.f90` and subroutine or function name `s1` is printed, with the line number of the accelerator region, which in the example is **15**.

- The library counts how many times the region is entered (1 in the example) and the microseconds spent in the region (in this example `1490738`), which is split into initialization time (in this example `1489138`) and execution time (in this example `1600`).
- The execution time is then divided into kernel execution time and data transfer time between the host and GPU.
- For each kernel, the line number is given, (`18` in the example), along with a count of kernel launches, and the total, maximum, minimum, and average time spent in the kernel, all of which are 155 in this example.

Related Accelerator Programming Tools

NVIDIA CUDA Profile

You can use the NVIDIA CUDA Profiler with PGI-generated code for the NVIDIA GPUs. You may download the CUDA Profiler from the same website as the CUDA software:

```
www.nvidia.com/cuda
```

Documentation and support is provided by NVIDIA.

TAU - Tuning and Analysis Utility

You can use the TAU (Tuning and Analysis Utilities), version 2.18.1+, with PGI-generated accelerator code. TAU instruments code at the function or loop level, and version 2.18.1 is enhanced with support to track performance in accelerator regions. TAU software and documentation is available at this website:

```
http://tau.uoregon.edu
```

Supported Intrinsics

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

Intrinsics make the use of processor-specific enhancements easier because they provide a language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

Supported Fortran Intrinsics Summary Table

Table 7.6 is an alphabetical summary of the supported Fortran intrinsics that the accelerator supports. These functions are specific to Fortran 90/95 unless otherwise specified.

Note

For complete descriptions of these intrinsics, refer to the Chapter 6, "Fortran Intrinsics" of the PGI Fortran Reference.

In most cases PGI provides support for all the data types for which the intrinsic is valid. When support is available for only certain data types, the middle column of the table specifies which ones, using these codes: I for integer, S for single precision real, D for double precision real, C for single precision complex, and Z for double precision complex.

Table 7.6. Supported Fortran Intrinsic Summary Table

This intrinsic		Returns this value ...
ABS	I,S,D	Determine the absolute value of the supplied argument.
ACOS		arccosine of the specified value.
AINT		Truncate the supplied value to a whole number.
ANINT		nearest whole number to the supplied argument.
ASIN		arcsine of the specified value.
ATAN		arctangent of the specified value.
ATAN2		arctangent of the specified value.
COS	S,D	cosine of the specified value.
COSH		hyperbolic cosine of the specified value.
DBLE	S,D	conversion of the value to double precision real.
DPROD		double precision real product.
EXP	S,D	exponential value of the argument.
IAND		Perform a bit-by-bit logical AND on the arguments.
IEOR		Perform a bit-by-bit logical exclusive OR on the arguments.
INT	I,S,D	Convert a value to integer type.
IOR		Perform a bit-by-bit logical OR on the arguments.
LOG	S,D	natural logarithm of the specified value.
LOG10		base-10 logarithm of the specified value.
MAX		maximum value of the supplied arguments.
MIN		minimum value of the supplied arguments.
MOD	I	Find the remainder.
NINT		nearest integer to the real argument.
NOT		Perform a bit-by-bit logical complement on the argument.
REAL	I,S,D	Convert the argument to real.
SIGN		absolute value of A times the sign of B.
SIN	S,D	value of the sine of the argument.
SINH		hyperbolic sine of the argument.
SQRT	S,D	square root of the argument.
TAN		tangent of the specified value.

This intrinsic		Returns this value ...
TANH		hyperbolic tangent of the specified value.

Supported C Intrinsics Summary Table

This section contains an alphabetical summary by category of the C intrinsics that the accelerator supports.

Double Functions

Table 7.7. Supported C Intrinsic Double Functions

This intrinsic	Returns this value ...
acos	arccosine of the specified value.
asin	arcsine of the specified value.
atan	arctangent of the specified value.
atan2	arctangent of y/x, where y is the first argument, x the second.
cos	cosine of the specified value.
cosh	hyperbolic cosine of the specified value.
exp	exponential value of the argument.
fabs	absolute value of the argument.
fmax	maximum value of the two supplied arguments
fmin	minimum value of the two supplied arguments
log	natural logarithm of the specified value.
log10	base-10 logarithm of the specified value.
pow	value of the first argument raised to the power of the second argument.
sin	value of the sine of the argument.
sinh	hyperbolic sine of the argument.
sqrt	square root of the argument.
tan	tangent of the specified value.
tanh	hyperbolic tangent of the specified value.

Float Functions

Table 7.8. Supported C Intrinsic Float Functions

This intrinsic	Returns this value ...
acosf	arccosine of the specified value.
asinf	arcsine of the specified value.
atanf	arctangent of the specified value.
atan2f	arctangent of y/x, where y is the first argument, x the second.

This intrinsic	Returns this value ...
cosf	cosine of the specified value.
coshf	hyperbolic cosine of the specified value.
expf	exponential value of the floating-point argument.
fabsf	absolute value of the floating-point argument.
logf	natural logarithm of the specified value.
log10f	base-10 logarithm of the specified value.
powf	value of the first argument raised to the power of the second argument.
sinf	value of the sine of the argument.
sinhf	hyperbolic sine of the argument.
sqrtf	square root of the argument.
tanf	tangent of the specified value.
tanhf	hyperbolic tangent of the specified value.

References related to Accelerators

- ISO/IEC 1539-1:1997, Information Technology - Programming Languages - Fortran, Geneva, 1997 (Fortran 95).
- American National Standard Programming Language C, ANSI X3.159-1989 (ANSI C).
- ISO/IEC 9899:1999, Information Technology - Programming Languages - C, Geneva, 1999 (C99).
- PGI Tools Guide, The Portland Group, Release 9.0, June, 2009. Available online at http://www.pgroup.com/doc/pgitools.pdf.
- PGI Fortran Reference, The Portland Group, Release 9.0, November, 2009. Available online at http://www.pgroup.com/doc/pgifortref.pdf

Chapter 8. Using Directives and Pragmas

It is often useful to be able to alter the effects of certain command line options or default behavior of the compiler. Fortran directives and C/C++ pragmas provide pragmatic information that control the actions of the compiler in a particular portion of a program without affecting the program as a whole. That is, while a command line option affects the entire source file that is being compiled, directives and pragmas apply, or disable, the effects of a command line option to selected subprograms or to selected loops in the source file, for example, to optimize a specific area of code. Use directives and pragmas to tune selected routines or loops.

PGI Proprietary Fortran Directives

PGI Fortran compilers support proprietary directives that may have any of the following forms:

```
!pgi$g directive
!pgi$r directive
!pgi$l directive
!pgi$ directive
```

Note

If the input is in fixed format, the comment character must begin in column 1 and either * or C is allowed in place of !.

The scope indicator controls the scope of the directive. This indicator occurs after the $. Some directives ignore the scope indicator.

The valid scopes, shown in the previous forms of the directive, are these:

g
: (global) indicates the directive applies to the end of the source file.

r
: (routine) indicates the directive applies to the next subprogram.

l
: (loop) indicates the directive applies to the next loop, but not to any loop contained within the loop body. Loop-scoped directives are only applied to `DO` loops.

blank
: indicates that the default scope for the directive is applied.

The body of the directive may immediately follow the scope indicator. Alternatively, any number of blanks may precede the name of the directive. Any names in the body of the directive, including the directive name, may not contain embedded blanks. Blanks may surround any special characters, such as a comma or an equal sign.

The directive name, including the directive prefix, may contain upper or lower case letters, and the case is not significant. Case is significant for any variable names that appear in the body of the directive if the command line option `-Mupcase` is selected. For compatibility with other vendors' directives, the prefix `cpgi$` may be substituted with `cdir$` or `cvd$`.

PGI Proprietary C and C++ Pragmas

Pragmas may be supplied in a C/C++ source file to provide information to the compiler. Many pragmas have a corresponding command-line option. Pragmas may also toggle an option, selectively enabling and disabling the option.

The general syntax of a pragma is:

```
#pragma [ scope ] pragma-body
```

The optional scope field is an indicator for the scope of the pragma; some pragmas ignore the scope indicator.

The valid scopes are:

global
: indicates the pragma applies to the entire source file.

routine
: indicates the pragma applies to the next function.

loop
: indicates the pragma applies to the next loop (but not to any loop contained within the loop body). Loop-scoped pragmas are only applied to for and while loops.

If a scope indicator is not present, the default scope, if any, is applied. Whitespace must appear after the pragma keyword and between the scope indicator and the body of the pragma. Whitespace may also surround any special characters, such as a comma or an equal sign. Case is significant for the names of the pragmas and any variable names that appear in the body of the pragma.

PGI Proprietary Optimization Directive and Pragma Summary

The following table summarizes the supported Fortran directives and C/C++ pragmas. The following terms are useful in understanding the table.

- Functionality is a brief summary of the way to use the directive or pragma. For a complete description, refer to Chapter 20, "*Directives and Pragmas Reference*," on page 327.
- Many of the directives and pragmas can be preceded by `NO`. The default entry indicates the default for the directive or pragma. N/A appears if a default does not apply.

- The scope entry indicates the allowed scope indicators for each directive or pragma, with `L` for loop, `R` for routine, and `G` for global. The default scope is surrounded by parentheses and N/A appears if the directive or pragma is not available in the given language.

  ### Note

  The "*" in the scope indicates this:

  For routine-scoped directive
  : The scope includes the code following the directive or pragma until the end of the routine.

  For globally-scoped directive
  : The scope includes the code following the directive or pragma until the end of the file rather than for the entire file.

  ### Note

  The name of a directive or pragma may also be prefixed with `-M`.

  For example, you can use the directive `-Mbounds`, which is equivalent to the directive `bounds` and you can use `-Mopt`, which is equivalent to `opt`. For pragmas, you can use the directive `-Mnoassoc`, which is equivalent to the pragma `noassoc`, and `-Mvintr`, which is equivalent to `vintr`.

Table 8.1. Proprietary Optimization-Related Fortran Directive and C/C++ Pragma Summary

Directive or pragma	Functionality	Default	Fortran Scope	C/C++ Scope
altcode (noaltcode)	Do/don't generate alternate code for vectorized and parallelized loops.	altcode	(L)RG	(L)RG
assoc (noassoc)	Do/don't perform associative transformations.	assoc	(L)RG	(L)RG
bounds (nobounds)	Do/don't perform array bounds checking.	nobounds	(R)G*	(R)G
cncall (nocncall)	Loops are considered for parallelization, even if they contain calls to user-defined subroutines or functions, or if their loop counts do not exceed usual thresholds.	nocncall	(L)RG	(L)RG
concur (noconcur)	Do/don't enable auto-concurrentization of loops.	concur	(L)RG	(L)RG
depchk (nodepchk)	Do/don't ignore potential data dependencies.	depchk	(L)RG	(L)RG
eqvchk (noeqvchk)	Do/don't check EQUIVALENCE for data dependencies.	eqvchk	(L)RG	N/A
fcon (nofcon)	Do/don't assume unsuffixed real constants are single precision.	nofcon	N/A	(R)G

Directive or pragma	Functionality	Default	Fortran Scope	C/C++ Scope
invarif (noinvarif)	Do/don't remove invariant if constructs from loops.	invarif	(L)RG	(L)RG
ivdep	Ignore potential data dependencies.	ivdep	(L)RG	N/A
lstval (nolstval)	Do/don't compute last values.	lstval	(L)RG	(L)RG
prefetch	Control how prefetch instructions are emitted			
opt	Select optimization level.	N/A	(R)G	(R)G
safe (nosafe)	Do/don't treat pointer arguments as safe.	safe	N/A	(R)G
safe_lastval	Parallelize when loop contains a scalar used outside of loop.	not enabled	(L)	(L)
safeptr (nosafeptr)	Do/don't ignore potential data dependencies to pointers.	nosafeptr	N/A	L(R)G
single (nosingle)	Do/don't convert float parameters to double.	nosingle	N/A	(R)G*
tp	Generate PGI Unified Binary code optimized for specified targets.	N/A	(R)G	(R)G
unroll (nounroll)	Do/don't unroll loops.	nounroll	(L)RG	(L)RG
vector (novector)	Do/don't perform vectorizations.	vector	(L)RG*	(L)RG
vintr (novintr)	Do/don't recognize vector intrinsics.	vintr	(L)RG	(L)RG

Scope of Fortran Directives and Command-Line options

During compilation the effect of a directive may be to either turn an option on, or turn an option off. Directives apply to the section of code following the directive, corresponding to the specified scope, which may include the following loop, the following routine, or the rest of the program. This section presents several examples that show the effect of directives as well as their scope.

Consider the following Fortran code:

```
integer maxtime, time
parameter (n = 1000, maxtime = 10)
double precision a(n,n), b(n,n), c(n,n)
  do time = 1, maxtime
    do i = 1, n
      do j = 1, n
        c(i,j) = a(i,j) + b(i,j)
      enddo
    enddo
  enddo
end
```

When compiled with `-Mvect`, both interior loops are interchanged with the outer loop.

```
$ pgf95 -Mvect dirvect1.f
```

Directives alter this behavior either globally or on a routine or loop by loop basis. To assure that vectorization is not applied, use the `novector` directive with global scope.

```
cpgi$g novector
 integer maxtime, time
 parameter (n = 1000, maxtime = 10)
 double precision a(n,n), b(n,n), c(n,n)
   do time = 1, maxtime
     do i = 1, n
       do j = 1, n
         c(i,j) = a(i,j) + b(i,j)
       enddo
     enddo
   enddo
 end
```

In this version, the compiler disables vectorization for the entire source file. Another use of the directive scoping mechanism turns an option on or off locally, either for a specific procedure or for a specific loop:

```
 integer maxtime, time
 parameter (n = 1000, maxtime = 10)
 double precision a(n,n), b(n,n), c(n,n)
cpgi$l novector
   do time = 1, maxtime
     do i = 1, n
       do j = 1, n
         c(i,j) = a(i,j) + b(i,j)
       enddo
     enddo
   enddo
 end
```

Loop level scoping does not apply to nested loops. That is, the directive only applies to the following loop. In this example, the directive turns off vector transformations for the top-level loop. If the outer loop were a timing loop, this would be a practical use for a loop-scoped directive.

Scope of C/C++ Pragmas and Command-Line Options

During compilation a pragma either turns an option on or turns an option off. Pragmas apply to the section of code corresponding to the specified scope - either the entire file, the following loop, or the following or current routine. This section presents several examples showing the effect of pragmas and the use of the pragma scope indicators.

Note

In all cases, pragmas override a corresponding command-line option.

For pragmas that have only routine and global scope, there are two rules for determining the scope of the pragma. We cover these special scope rules at the end of this section.

Consider the following program:

```
main() {
   float a[100][100], b[100][100], c[100][100];
   int time, maxtime, n, i, j;
   maxtime=10;
```

```
   n=100;
   for (time=0; time<maxtime;time++)
     for (j=0; j<n;j++)
       for (i=0; i<n;i++)
         c[i][j] = a[i][j] + b[i][j];
}
```

When this is compiled using the `-Mvect` command-line option, both interior loops are interchanged with the outer loop. Pragmas alter this behavior either globally or on a routine or loop by loop basis. To ensure that vectorization is not applied, use the `novector` pragma with global scope.

```
main() {
#pragma global novector
   float a[100][100], b[100][100],c[100][100];
   int time, maxtime, n, i, j;
   maxtime=10;
   n=100;
   for (time=0; time<maxtime;time++)
     for (j=0; j<n;j++)
       for (i=0; i<n;i++)
         c[i][j] = a[i][j] + b[i][j];
}
```

In this version, the compiler does not perform vectorization for the entire source file. Another use of the pragma scoping mechanism turns an option on or off locally either for a specific procedure or for a specific loop. The following example shows the use of a loop-scoped pragma.

```
main() {
   float a[100][100], b[100][100],c[100][100];
   int time, maxtime, n, i, j;
   maxtime=10;
   n=100;
#pragma loop novector
   for (time=0; time<maxtime;time++)
     for (j=0; j<n;j++)
       for (i=0; i<n;i++)
         c[i][j] = a[i][j] + b[i][j];
}
```

Loop level scoping does not apply to nested loops. That is, the pragma only applies to the following loop. In this example, the pragma turns off vector transformations for the top-level loop. If the outer loop were a timing loop, this would be a practical use for a loop-scoped pragma. The following example shows routine pragma scope:

```
#include "math.h"
func1() {
#pragma routine novector
 float a[100][100], b[100][100];
 float c[100][100], d[100][100];
 int i,j;
 for (i=0;i<100;i++)
   for (j=0;j<100;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
func2() {
 float a[200][200], b[200][200];
 float c[200][200], d[200][200];
 int i,j;
```

```
 for (i=0;i<200;i++)
   for (j=0;j<200;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
```

When this source is compiled using the `-Mvect` command-line option, func2 is vectorized but func1 is not vectorized. In the following example, the global novector pragma turns off vectorization for the entire file.

```
#include "math.h"
func1() {
#pragma global novector
 float a[100][100], b[100][100];
 float c[100][100], d[100][100];
 int i,j;
 for (i=0;i<100;i++)
   for (j=0;j<100;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
func2() {
 float a[200][200], b[200][200];
 float c[200][200], d[200][200];
 int i,j;
 for (i=0;i<200;i++)
   for (j=0;j<200;j++)
     a[i][j] = a[i][j] + b[i][j] * c[i][j];
     c[i][j] = c[i][j] + b[i][j] * d[i][j];
}
```

Special Scope Rules

Special rules apply for a pragma with loop, routine, and global scope. When the pragma is placed within a routine, it applies to the routine from its point in the routine to the end of the routine. The same rule applies for one of these pragmas with global scope.

However, there are several pragmas for which only routine and global scope applies and which affect code immediately following the pragma:

- bounds and fcon – The bounds and fcon pragmas behave in a similar manner to pragmas with loop scope. That is, they apply to the code following the pragma.
- opt and safe – When the opt, and safe pragmas are placed within a routine, they apply to the entire routine as if they had been placed at the beginning of the routine.

Prefetch Directives and Pragmas

Today's processors are so fast that it is difficult to bring data into them quickly enough to keep them busy. Prefetch instructions can increase the speed of an application substantially by bringing data into cache so that it is available when the processor needs it.

When vectorization is enabled using the `-Mvect` or `-Mprefetch` compiler options, or an aggregate option such as `-fast` that incorporates `-Mvect`, the PGI compilers selectively emit instructions to explicitly prefetch data into the data cache prior to first use. You can control how these prefetch instructions are emitted by using prefetch directives and pragmas.

For a list of processors that support prefetch instructions refer to the PGI Release Notes.

Prefetch Directive Syntax

The syntax of a prefetch directive is as follows:

```
  c$mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

Prefetch Directive Format Requirements

Note

The sentinel for prefetch directives is `c$mem`, which is distinct from the `cpgi$` sentinel used for optimization directives. Any prefetch directives that use the `cpgi$` sentinel are ignored by the PGI compilers.

- The "c" must be in column 1.
- Either * or ! is allowed in place of c.
- The scope indicators g, r and l used with the `cpgi$` sentinel are not supported.
- The directive name, including the directive prefix, may contain upper or lower case letters and is case insensitive (case is not significant).
- If the command line option `-Mupcase` is used, any variable names that appear in the body of the directive are case sensitive.

Sample Usage of Prefetch Directive

Example 8.1. Prefetch Directive Use

This example uses prefetch directives to prefetch data in a matrix multiplication inner loop where a row of one source matrix has been gathered into a contiguous vector.

```
 real*8 a(m,n), b(n,p), c(m,p), arow(n)
...
 do j = 1, p
c$mem prefetch arow(1),b(1,j)
c$mem prefetch arow(5),b(5,j)
c$mem prefetch arow(9),b(9,j)
 do k = 1, n, 4
c$mem prefetch arow(k+12),b(k+12,j)
 c(i,j) = c(i,j) + arow(k) * b(k,j)
 c(i,j) = c(i,j) + arow(k+1) * b(k+1,j)
 c(i,j) = c(i,j) + arow(k+2) * b(k+2,j)
 c(i,j) = c(i,j) + arow(k+3) * b(k+3,j)
 enddo
 enddo
```

This pattern of prefetch directives the compiler emits prefetch instructions whereby elements of `arow` and `b` are fetched into the data cache starting four iterations prior to first use. By varying the prefetch distance in this way, it is sometimes possible to reduce the effects of main memory latency and improve performance.

Prefetch Pragma Syntax

The syntax of a prefetch pragma is as follows:

```
#pragma mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

Sample Usage of Prefetch Pragma

Example 8.2. Prefetch Pragma in C

This example uses the prefetch pragma to prefetch data from the source vector x for eight iterations beyond the current iteration.

```
 for (i=0; i<n; i++)  {
   #pragma mem prefetch x[i+8]
   y[i] = y[i] + a*x[i];
}
```

!DEC$ Directives

PGI Fortran compilers for Microsoft Windows support several de-facto standard Fortran directives that help with inter-language calling and importing and exporting routines to and from DLLs. These directives all take the form:

```
!DEC$ directive
```

Format Requirements

You must follow the following format requirements for the directive to be recognized in your program:

- The directive must begin in line 1 when the file is fixed format or compiled with `-Mfixed`.
- The directive prefix `!DEC$` requires a space between the prefix and the directive keyword, such as `ATTRIBUTES`.
- The ! must begin the prefix when compiling Fortran 90/95 free-form format.
- The characters C or * can be used in place of ! in either form of the prefix when compiling F77-style fixed-form format.
- The directives are completely case insensitive.

Summary Table

The following table summarizes the supported !DEC$ directives. For a complete description of each directive, refer to the section "!DEC$ Directives," on page 334 in Chapter 20, "*Directives and Pragmas Reference*".

Table 8.2. !DEC$ Directives Summary Table

Directive	Functionality
ALIAS	Specifies an alternative name with which to resolve a routine.
ATTRIBUTES	Lets you specify properties for data objects and procedures.
DECORATE	Specifies that the name specified in the ALIAS directive should have the prefix and postfix decorations performed on it that are associated with the calling conventions that are in effect. This directive has no effect if ALIAS is not specified.
DISTRIBUTE	Tells the compiler at what point within a loop to split into two loops.
IGNORE_TKR	Directs the compiler to ignore the type, kind, and/or rank (/TKR/) of specified dummy arguments in a procedure interface.

C$PRAGMA C

When programs are compiled using one of the PGI Fortran compilers on Linux, Win64, OSX, and SUA systems, an underscore is appended to Fortran global names, including names of functions, subroutines, and common blocks. This mechanism distinguishes Fortran name space from C/C++ name space.

You can use C$PRAGMA C in the Fortran program to call a C/C++ function from Fortran. The statement would look similar to this:

```
C$PRAGMA C(name[,name]...)
```

NOTE

This statement directs the compiler to recognize the routine 'name' as a C function, thus preventing the Fortran compiler from appending an underscore to the routine name.

On Win32 systems the C$PRAGMA C as well as the attributes C and STDCALL may effect other changes on argument passing as well as on the names of the routine. For more information on this topic, refer to "Win32 Calling Conventions," on page 158.

Chapter 9. Creating and Using Libraries

A library is a collection of functions or subprograms that are grouped for reference and ease of linking. This chapter discusses issues related to PGI-supplied compiler libraries. Specifically, it addresses the use of C/C++ builtin functions in place of the corresponding libc routines, creation of dynamically linked libraries, known as shared objects or shared libraries, and math libraries.

> **Note**
>
> This chapter does not duplicate material related to using libraries for inlining, described in "Creating an Inline Library," on page 51 or information related to run-time library routines available to OpenMP programmers, described in "Run-time Library Routines," on page 65.

PGI provides libraries that export C interfaces by using Fortran modules. It also provides additions to the supported library functionality, specifically, NARGS, a run-time function included in DFLIB. NARGS returns the total number of command-line arguments, including the command. The result is of type INTEGER(4). For example, NARGS returns 4 for the command-line invocation of PROG1 -g -c -a.

This chapter has examples that include the following options related to creating and using libraries.

`-Bdynamic`	`-def<file>`	`-implib <file>`	`-Mmakeimplib`
`-Bstatic`	`-dynamiclib`	`-l`	`-o`
`-c`	`-fpic`	`-Mmakedll`	`-shared`

Using builtin Math Functions in C/C++

The name of the math header file is `math.h`. Include the math header file in all of your source files that use a math library routine as in the following example, which calculates the inverse cosine of pi/3.

```
#include <math.h>
#define PI 3.1415926535
void main()
```

```
{
  double x, y;
  x = PI/3.0;
  y = acos(x);
}
```

Including `math.h` causes PGCC C and C++ to use builtin functions, which are much more efficient than library calls. In particular, if you include `math.h`, the following intrinsics calls are processed using builtins:

abs	atan	atan2	cos
exp	fabs	fmax	fmaxf
fmin	fminf	log	log10
pow	sin	sqrt	tan

Using System Library Routines

Release 9.0 of the PGI run-time libraries makes use of Linux system libraries to implement, for example, OpenMP and Fortran I/O. The PGI run-time libraries make use of several additional system library routines.

On 64-bit Linux systems, the system library routines that PGI supports include these:

aio_error	aio_write	pthread_mutex_init	sleep
aio_read	calloc	pthread_mutex_lock	
aio_return	getrlimit	pthread_mutex_unlock	
aio_suspend	pthread_attr_init	setrlimit	

On 32-bit Linux systems, the system library routines that PGI supports include these:

aio_error	aio_suspend	getrlimit	sleep
aio_read	aio_write	pthread_attr_init	
aio_return	calloc	setrlimit	

Creating and Using Shared Object Files on Linux

All of the PGI Fortran, C, and C++ compilers support creation of shared object files. Unlike statically-linked object and library files, shared object files link and resolve references with an executable at runtime via a dynamic linker supplied with your operating system. The PGI compilers must generate position independent code to support creation of shared objects by the linker. However, this is not the default. You must create object files with position independent code and shared object files that will include them.

Procedure to create a use a shared object file

The following steps describe how to create and use a shared object file.

1. Create an object file with position independent code.

   To do this, compile your code with the appropriate PGI compiler using the `-fpic` option, or one of the equivalent options, such as `-fPIC`, `-Kpic,` and `-KPIC`, which are supported for compatibility with

other systems. For example, use the following command to create an object file with position independent code using pgf95:

```
% pgf95 -c -fpic tobeshared.f
```

2. Produce a shared object file.

   To do this, use the appropriate PGI compiler to invoke the linker supplied with your system. It is customary to name such files using a `.so` filename extension. On Linux, you do this by passing the `-shared` option to the linker:

   ```
   % pgf95 -shared -o tobeshared.so tobeshared.o
   ```

   Note

   Compilation and generation of the shared object can be performed in one step using both the `-fpic` option and the appropriate option for generation of a shared object file.

3. Use a shared object file.

   To do this, use the appropriate PGI compiler to compile and link the program which will reference functions or subroutines in the shared object file, and list the shared object on the link line, as shown here:

   ```
   % pgf95 -o myprog myprog.f tobeshared.so
   ```

4. Make the executable available.

   You now have an executable `myprog` which does not include any code from functions or subroutines in `tobeshared.so`, but which can be executed and dynamically linked to that code. By default, when the program is linked to produce `myprog`, no assumptions are made on the location of `tobeshared.so`. Therefore, for `myprog` to execute correctly, you must initialize the environment variable `LD_LIBRARY_PATH` to include the directory containing `tobeshared.so`. If `LD_LIBRARY_PATH` is already initialized, it is important not to overwrite its contents. Assuming you have placed `tobeshared.so` in a directory `/home/myusername/bin`, you can initialize `LD_LIBRARY_PATH` to include that directory and preserve its existing contents, as shown in the following:

   ```
   % setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":/home/myusername/bin
   ```

   If you know that `tobeshared.so` always resides in a specific directory, you can create the executable `myprog` in a form that assumes this directory by using the `-R` link-time option. For example, you can link as follows:

   ```
   % pgf95 -o myprog myprof.f tobeshared.so -R/home/myusername/bin
   ```

   Note

   As with the `-L` option, there is no space between `-R` and the directory name. If the –R option is used, it is not necessary to initialize `LD_LIBRARY_PATH`.

   In the previous example, the dynamic linker always looks in `/home/myusername/bin` to resolve references to `tobeshared.so`. By default, if the `LD_LIBRARY_PATH` environment variable is not set, the linker only searches `/usr/lib` and `/lib` for shared objects.

ldd Command

The `ldd` command is a useful tool when working with shared object files and executables that reference them. When applied to an executable, as shown in the following example, `ldd` lists all shared object files referenced in the executable along with the pathname of the directory from which they will be extracted.

```
% ldd myprog
```

If the pathname is not hard-coded using the `-R` option, and if `LD_LIBRARY_PATH` is not initialized, the pathname is listed as "not found". For more information on `ldd`, its options and usage, see the online man page for `ldd`.

Creating and Using Shared Object Files in SFU and 32-bit SUA

> **Note**
>
> The information included in this section is valid for 32-bit only.

The 32-bit version of PGI Workstation for SFU and SUA uses the GNU `ld` for its linker, unlike previous versions that used the Windows `LINK.EXE`. With this change, the PGI compilers and tools for SFU and 32-bit SUA are now able to generate shared object (`.so`) files. You use the `-shared` switch to generate a shared object file.

The following example creates a shared object file, `hello.so`, and then creates a program called `hello` that uses it.

1. Create a shared object file.

   To produce a shared object file, use the appropriate PGI compiler to invoke the linker supplied with your system. It is customary to name such files using a `.so` filename extension. In the following example, we use `hello.so`:

   ```
   % pgcc -shared hello.c -o hello.so
   ```

2. Create a program that uses the shared object, in this example, `hello.so`:

   ```
   % pgcc hi.c hello.so -o hello
   ```

Shared Object Error Message

When running a program that uses a shared object, you may encounter an error message similar to the following:

```
hello: error in loading shared libraries hello.so:
 cannot open shared object file: No such file or directory
```

This error message either means that the shared object file does not exist or that the location of this file is not specified in your `LD_LIBRARY_PATH` variable. To specify the location of the `.so`, add the shared object's directory to your `LD_LIBRARY_PATH` variable. For example, the following command adds the current directory to your `LD_LIBRARY_PATH` variable using C shell syntax:

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":"./"
```

Shared Object-Related Compiler Switches

The following switches support shared object files in SFU and SUA. For more detailed information on these switches, refer to Chapter 16, "*Command-Line Options Reference*," on page 195.

–shared
Used to produce shared libraries

–Bdynamic
Passed to linker; specify dynamic binding

> **Note**
>
> On Windows, -Bstatic and -Bdynamic must be used for *both* compiling and linking.

–Bstatic
Passed to linker; specify static binding

–Bstatic_pgi
Use to link static PGI libraries with dynamic system libraries; implies `–Mnorpath`.

–L<libdir>
Passed to linker; add directory to library search path.

–Mnorpath
Don't add `–rpath` paths to link line.

–Mnostartup
Do not use standard linker startup file.

–Mnostdlib
Do not use standard linker libraries.

–R<ldarg>
Passed to linker; just link symbols from object, or add directory to run time search path.

Creating and Using Dynamic Libraries on Mac OS X

> **Note**
>
> PGI compilers for Mac OS X do not support static linking on user executables. Apple only ships dynamic versions of its system libraries - not static versions. You can create static libraries; however, you cannot create 100% static executables.

The 32-bit version of PGI Workstation for Mac OS X supports generation of dynamic libraries. To create the dynamic library, you use the `–dynamiclib` switch to invoke the libtool utility program provided by Mac OS X. For more information, refer to the `libtool` man page.

The following example creates and uses a dynamic library:

1. Create the object files.

   `world.f90:`

```
subroutine world
print *, 'Hello World!'
end
```

```
hello.f90:

program hello
call world
end
```

2. Build the dynamic library:

```
% pgf95 -dynamiclib world.f90 -o world.dylib
```

3. Build the program that uses the dynamic library:

```
% pgf95 hello.f90 world.dylib -o hello
```

4. Run the program:

```
% ./hello|
 Hello World!
```

PGI Runtime Libraries on Windows

The PGI runtime libraries on Windows are available in both static and dynamically-linked (DLL) versions. The static libraries are used by default.

- You can use the dynamically-linked version of the run-time by specifying `-Bdynamic` at both compile and link time.
- You can explicitly specify static linking, the default, by using `-Bstatic` at compile and link time.

For details on why you might choose one type of linking over another type, refer to "Creating and Using Dynamic-Link Libraries on Windows," on page 118.

Creating and Using Static Libraries on Windows

The Microsoft Library Manager (`LIB.EXE`) is the tool that is typically used to create and manage a static library of object files on Windows. `LIB` is provided with the PGI compilers as part of the Microsoft Open Tools. Refer to *www.msdn2.com* for a complete `LIB` reference - search for `LIB.EXE`. For a list of available options, invoke `LIB` with the `/?` switch.

For compatibility with legacy makefiles, PGI provides a wrapper for `LIB` and `LINK` called `ar`. This version of `ar` is compatible with Windows and object-file formats.

PGI also provides `ranlib` as a placeholder for legacy makefile support.

ar command

The `ar` command is a legacy archive wrapper that interprets legacy `ar` command line options and translates these to `LINK/LIB` options. You can use it to create libraries of object files.

Syntax:

The syntax for the `ar` command is this:

```
ar [options] [archive] [object file].
```

Where:

- The first argument must be a command line switch, and the leading dash on the first option is optional.
- The single character options, such as `-d` and `-v`, may be combined into a single option, as `-dv`.

  Thus, `ar dv`, `ar -dv,` and `ar -d -v` all mean the same thing.
- The first non-switch argument must be the library name.
- One (and only one) of `-d`, `-r`, `-t`, or `-x` must appear on the command line.

Options

The options available for the `ar` command are these:

–c
: This switch is for compatibility; it is ignored.

–d
: Deletes the named object files from the library.

–r
: Replaces in or adds the named object files to the library.

–t
: Writes a table of contents of the library to standard out.

–v
: Writes a verbose file-by-file description of the making of the new library to standard out.

–x
: Extracts the named files by copying them into the current directory.

ranlib command

The `ranlib` command is a wrapper that allows use of legacy scripts and makefiles that use the `ranlib` command. The command actually does nothing; it merely exists for compatibility.

Syntax:

The syntax for the `ranlib` command is this:

```
ranlib [options] [archive]
```

Options

The options available for the `ranlib` command are these:

–help
: Short help information is printed out.

–V
: Version information is printed out.

Creating and Using Dynamic-Link Libraries on Windows

There are several differences between static and dynamic-link libraries on Windows. Libraries of either type are used when resolving external references for linking an executable, but the process differs for each type of library. When linking with a static library, the code needed from the library is incorporated into the executable. When linking with a DLL, external references are resolved using the DLL's import library, not the DLL itself. The code in the DLL associated with the external references does not become a part of the executable. The DLL is loaded when the executable that needs it is run. For the DLL to be loaded in this manner, the DLL must be in your path.

Static libraries and DLLs also handle global data differently. Global data in static libraries is automatically accessible to other objects linked into an executable. Global data in a DLL can only be accessed from outside the DLL if the DLL exports the data and the image that uses the data imports it.

To access global data, the C compilers support the Microsoft storage class extensions: `__declspec(dllimport)` and `__declspec(dllexport)`. These extensions may appear as storage class modifiers and enable functions and data to be imported and exported:

```
extern int __declspec(dllimport) intfunc();
float __declspec(dllexport) fdata;
```

The PGI Fortran compilers support the DEC$ ATTRIBUTES extensions `DLLIMPORT` and `DLLEXPORT`:

```
cDEC$ ATTRIBUTES DLLEXPORT :: object [,object] ...
cDEC$ ATTRIBUTES DLLIMPORT :: object [,object] ...
```

Here *c* is one of C, c, !, or *. `object` is the name of the subprogram or common block that is exported or imported. Further, common block names are enclosed within slashes (/), as shown here:

```
cDEC$ ATTRIBUTES DLLIMPORT :: intfunc
!DEC$ ATTRIBUTES DLLEXPORT :: /fdata/
```

For more information on these extensions, refer to "!DEC$ Directives," on page 109.

The examples in this section further illustrate the use of these extensions.

To create a DLL from the command line, use the `-Mmakedll` option.

The following switches apply to making and using DLLs with the PGI compilers:

–Bdynamic
: Compile for and link to the DLL version of the PGI runtime libraries. This flag is required when linking with any DLL built by the PGI compilers. This flag corresponds to the `/MD` flag used by Microsoft's `cl` compilers.

  When you use the PGI compiler flag `-Bdynamic` to create an executable that links to the DLL form of the runtime, the executable built is smaller than one built without `-Bdynamic`. The PGI runtime DLLs, however, must be available on the system where the executable is run. You must use the `-Bdynamic` flag when linking an executable against a DLL built by the PGI compilers.

–Bstatic
: Compile for and link to the static version of the PGI runtime libraries. This flag corresponds to the `/MT` flag used by Microsoft's `cl` compilers.

On Windows, you must use `-Bstatic` for both compiling and linking.

–Mmakedll
: Generate a dynamic-link library or DLL. Implies `-Bdynamic`.

–Mmakeimplib
: Generate an import library without generating a DLL. Use this flag when you want to generate an import library for a DLL but are not yet ready to build the DLL itself. This situation might arise, for example, when building DLLs with mutual imports, as shown in Example 9.4, "Build DLLs Containing Mutual Imports: Fortran," on page 122.

–o <file>
: Passed to the linker. Name the DLL or import library <file>.

–def <file>
: When used with `-Mmakedll`, this flag is passed to the linker and a `.def` file named <file> is generated for the DLL. The `.def` file contains the symbols exported by the DLL. Generating a `.def` file is not required when building a DLL but can be a useful debugging tool if the DLL does not contain the symbols that you expect it to contain.

  When used with `-Mmakeimplib`, this flag is passed to `lib` which requires a `.def` file to create an import library. The `.def` file can be empty if the list of symbols to export are passed to `lib` on the command line or explicitly marked as `DLLEXPORT` in the source code.

–implib <file>
: Passed to the colinker. Generate an import library named <file> for the DLL. A DLL's import library is the interface used when linking an executable that depends on routines in a DLL.

To use the PGI compilers to create an executable that links to the DLL form of the runtime, use the compiler flag `-Bdynamic`. The executable built will be smaller than one built without `-Bdynamic`; the PGI runtime DLLs, however, must be available on the system where the executable is run. The `-Bdynamic` flag must be used when an executable is linked against a DLL built by the PGI compilers.

The following examples outline how to use `-Bdynamic`, `-Mmakedll` and `-Mmakeimplib` to build and use DLLs with the PGI compilers.

Example 9.1. Build a DLL: Fortran

This example builds a DLL from a single source file, `object1.f`, which exports data and a subroutine using `DLLEXPORT`. The main source file, `prog1.f`, uses `DLLIMPORT` to import the data and subroutine from the DLL.

`object1.f`

```
 subroutine sub1(i)
!DEC$ ATTRIBUTES DLLEXPORT :: sub1
 integer i
 common /acommon/ adata
 integer adata
!DEC$ ATTRIBUTES DLLEXPORT :: /acommon/
 print *, "sub1 adata", adata
 print *, "sub1 i ", i
 adata = i
```

```
 end
```

prog1.f

```
 program prog1
 common /acommon/ adata
 integer adata
 external sub1
!DEC$ ATTRIBUTES DLLIMPORT:: sub1, /acommon/
 adata = 11
 call sub1(12)
 print *, "main adata", adata
 end
```

Step 1: Create the DLL `obj1.dll` and its import library `obj1.lib` using the following series of commands:

```
% pgf95 -Bdynamic -c object1.f
% pgf95 -Mmakedll object1.obj -o obj1.dll
```

Step 2: Compile the main program:

```
% pgf95 -Bdynamic -o prog1 prog1.f -defaultlib:obj1
```

The `–Bdynamic` and `–Mmakedll` switches cause the compiler to link against the PGI runtime DLLs instead of the PGI runtime static libraries. The `–Bdynamic` switch is required when linking against any PGI-compiled DLL, such as `obj1.dll`. The -defaultlib: switch specifies that `obj1.lib`, the DLL's import library, should be used to resolve imports.

Step 3: Ensure that `obj1.dll` is in your path, then run the executable `prog1` to determine if the DLL was successfully created and linked:

```
% prog1
sub1 adata 11
sub1 i 12
main adata 12
```

Should you wish to change `obj1.dll` without changing the subroutine or function interfaces, no rebuilding of `prog1` is necessary. Just recreate `obj1.dll` and the new `obj1.dll` is loaded at runtime.

Example 9.2. Build a DLL: C

In this example, we build a DLL out of a single source file, `object2.c`, which exports data and a subroutine using `__declspec(dllexport)`. The main source file, `prog2.c`, uses `__declspec(dllimport)` to import the data and subroutine from the DLL.

object2.c

```
int __declspec(dllexport) data;
void __declspec(dllexport)
func2(int i)
{
  printf("func2: data == %d\n", data);
  printf("func2: i == %d\n", i);
  data = i;
}
```

prog2.c

```
int __declspec(dllimport) data;
```

```
void __declspec(dllimport) func2(int);
int
main()
{
  data = 11;
  func2(12);
  printf("main: data == %d\n",data);
  return 0;
}
```

Step 1: Create the DLL `obj2.dll` and its import library `obj2.lib` using the following series of commands:

```
% pgcc -Bdynamic -c object2.c
% pgcc -Mmakedll object2.obj -o obj2.dll
```

Step 2: Compile the main program:

```
% pgcc -Bdynamic -o prog2 prog2.c -defaultlib:obj2
```

The `-Bdynamic` switch causes the compiler to link against the PGI runtime DLLs instead of the PGI runtime static libraries. The `-Bdynamic` switch is required when linking against any PGI-compiled DLL such as `obj2.dll`. The `#defaultlib:` switch specifies that `obj2.lib`, the DLL's import library, should be used to resolve the imported data and subroutine in `prog2.c`.

Step 3: Ensure that `obj2.dll` is in your path, then run the executable `prog2` to determine if the DLL was successfully created and linked:

```
% prog2
func2: data == 11
func2: i == 12
main: data == 12
```

Should you wish to change `obj2.dll` without changing the subroutine or function interfaces, no rebuilding of `prog2` is necessary. Just recreate `obj2.dll` and the new `obj2.dll` is loaded at runtime.

Example 9.3. Build DLLs Containing Circular Mutual Imports: C

In this example we build two DLLs, `obj3.dll` and `obj4.dll`, each of which imports a routine that is exported by the other. To link the first DLL, the import library for the second DLL must be available. Usually an import library is created when a DLL is linked. In this case, however, the second DLL cannot be linked without the import library for the first DLL. When such circular imports exist, an import library for one of the DLLs must be created in a separate step without creating the DLL. The PGI drivers call the Microsoft `lib` tool to create import libraries in this situation. Once the DLLs are built, we can use them to build the main program.

```
/* object3.c */

 void __declspec(dllimport) func_4b(void);
 void __declspec(dllexport)
 func_3a(void)
 {
   printf("func_3a, calling a routine in obj4.dll\n");
   func_4b();
 }
 void __declspec(dllexport)
 func_3b(void)
 {
   printf("func_3b\n");
 }
```

```
/* object4.c */
 void __declspec(dllimport) func_3b(void);
 void __declspec(dllexport)
 func_4a(void)
 {
   printf("func_4a, calling a routine in obj3.dll\n");
   func_3b();
 }
 void __declspec(dllexport)
 func_4b(void)
 {
   printf("func_4b\n");
 }
```

```
/* prog3.c */
 void __declspec(dllimport) func_3a(void);
 void __declspec(dllimport) func_4a(void);
 int
 main()
 {
   func_3a();
   func_4a();
   return 0;
 }
```

Step 1: Use `-Mmakeimplib` with the PGI compilers to build an import library for the first DLL without building the DLL itself.

```
% pgcc -Bdynamic -c object3.c
% pgcc -Mmakeimplib -o obj3.lib object3.obj
```

The `-def=<deffile>` option can also be used with `–Mmakeimplib`. Use a `.def` file when you need to export additional symbols from the DLL. A `.def` file is not needed in this example because all symbols are exported using `__declspec(dllexport)`.

Step 2: Use the import library, `obj3.lib`, created in Step 1, to link the second DLL.

```
% pgcc -Bdynamic -c object4.c
% pgcc -Mmakedll -o obj4.dll object4.obj -defaultlib:obj3
```

Step 3: Use the import library, `obj4.lib`, created in Step 2, to link the first DLL.

```
% pgcc -Mmakedll -o obj3.dll object3.obj -defaultlib:obj4
```

Step 4: Compile the main program and link against the import libraries for the two DLLs.

```
% pgcc -Bdynamic prog3.c -o prog3 -defaultlib:obj3 -defaultlib:obj4
```

Step 5: Execute `prog3.exe` to ensure that the DLLs were create properly.

```
% prog3
func_3a, calling a routine in obj4.dll
func_4b
func_4a, calling a routine in obj3.dll
func_3b
```

Example 9.4. Build DLLs Containing Mutual Imports: Fortran

In this example we build two DLLs when each DLL is dependent on the other, and use them to build the main program. In the following source files, `object2.f95` makes calls to routines defined in `object3.f95`,

and vice versa. This situation of mutual imports requires two steps to build each DLL. To link the first DLL, the import library for the second DLL must be available. Usually an import library is created when a DLL is linked. In this case, however, the second DLL cannot be linked without the import library for the first DLL. When such circular imports exist, an import library for one of the DLLs must be created in a separate step without creating the DLL. The PGI drivers call the Microsoft `lib` tool to create import libraries in this situation. Once the DLLs are built, we can use them to build the main program.

`object2.f95`

```
 subroutine func_2a
 external func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3b
 print*,"func_2a, calling a routine in obj3.dll"
 call func_3b()
 end subroutine
 subroutine func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2b
 print*,"func_2b"
 end subroutine
```

`object3.f95`

```
 subroutine func_3a
 external func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2b
 print*,"func_3a, calling a routine in obj2.dll"
 call func_2b()
 end subroutine
```

```
 subroutine func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_3b
 print*,"func_3b"
 end subroutine
```

`prog2.f95`

```
 program prog2
 external func_2a
 external func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3a
 call func_2a()
 call func_3a()
 end program
```

Step 1: Use `–Mmakeimplib` with the PGI compilers to build an import library for the first DLL without building the DLL itself.

```
% pgf95 -Bdynamic -c object2.f95
% pgf95 -Mmakeimplib -o obj2.lib object2.obj
```

Tip

The `-def=<deffile>` option can also be used with `–Mmakeimplib`. Use a `.def` file when you need to export additional symbols from the DLL. A `.def` file is not needed in this example because all symbols are exported using `DLLEXPORT`.

Step 2: Use the import library, `obj2.lib`, created in Step 1, to link the second DLL.

```
% pgf95 -Bdynamic -c object3.f95
% pgf95 -Mmakedll -o obj3.dll object3.obj -defaultlib:obj2
```

Step 3: Use the import library, `obj3.lib`, created in Step 2, to link the first DLL.

```
% pgf95 -Mmakedll -o obj2.dll object2.obj -defaultlib:obj3
```

Step 4: Compile the main program and link against the import libraries for the two DLLs.

```
% pgf95 -Bdynamic prog2.f95 -o prog2 -defaultlib:obj2 -defaultlib:obj3
```

Step 5: Execute `prog2` to ensure that the DLLs were created properly:

```
% prog2
func_2a, calling a routine in obj3.dll
func_3b
func_3a, calling a routine in obj2.dll
func_2b
```

Example 9.5. Import a Fortran module from a DLL

In this example we import a Fortran module from a DLL. We use the source file `defmod.f90` to create a DLL containing a Fortran module. We then use the source file `use_mod.f90` to build a program that imports and uses the Fortran module from `defmod.f90`.

defmod.f90

```
module testm
 type a_type
 integer :: an_int
 end type a_type
 type(a_type) :: a, b
!DEC$ ATTRIBUTES DLLEXPORT :: a,b
 contains
 subroutine print_a
!DEC$ ATTRIBUTES DLLEXPORT :: print_a
 write(*,*) a%an_int
 end subroutine
 subroutine print_b
!DEC$ ATTRIBUTES DLLEXPORT :: print_b
 write(*,*) b%an_int
 end subroutine
end module
```

usemod.f90

```
use testm
 a%an_int = 1
 b%an_int = 2
 call print_a
 call print_b
end
```

Step 1: Create the DLL.

```
% pgf90 -Mmakedll -o defmod.dll defmod.f90
Creating library defmod.lib and object defmod.exp
```

Step 2: Create the `exe` and link against the import library for the imported DLL.

```
% pgf90 -Bdynamic -o usemod usemod.f90 -defaultlib:defmod.lib
```

Step 3: Run the `exe` to ensure that the module was imported from the DLL properly.

```
% usemod
1
2
```

Using LIB3F

The PGI Fortran compilers include complete support for the de facto standard LIB3F library routines on both Linux and Windows operating systems. See the PGI Fortran Reference manual for a complete list of available routines in the PGI implementation of LIB3F.

LAPACK, BLAS and FFTs

Pre-compiled versions of the public domain LAPACK and BLAS libraries are included with the PGI compilers. The LAPACK library is called `liblapack.a` or on Windows, `liblapack.lib`. The BLAS library is called `libblas.a` or on Windows, `libblas.lib`. These libraries are installed to `$PGI/<target>/lib`, where <target> is replaced with the appropriate target name (linux86, linux86-64, osx86, osx86-64, win32, win64, sfu32, sua32, or sua64).

To use these libraries, simply link them in using the `-l` option when linking your main program:

```
% pgf95 myprog.f -llapack -lblas
```

Highly optimized assembly-coded versions of BLAS and certain FFT routines may be available for your platform. In some cases, these are shipped with the PGI compilers. See the current release notes for the PGI compilers you are using to determine if these optimized libraries exist, where they can be downloaded (if necessary), and how to incorporate them into your installation as the default.

The C++ Standard Template Library

The PGC++ compiler includes a bundled copy of the STLPort Standard C++ Library. See the online Standard C ++ Library tutorial and reference manual at *www.stlport.com* for further details and licensing.

Chapter 10. Using Environment Variables

Environment variables allow you to set and pass information that can alter the default behavior of the PGI compilers and the executables which they generate. This chapter includes explanations of the environment variables specific to PGI compilers. Other environment variables are referenced and documented in other sections of this User's Guide or the PGI Tools Guide.

- You use OpenMP environment variables to control the behavior of OpenMP programs. For consistency related to the OpenMP environment, the details of the OpenMP-related environment variables are included in Chapter 5, "*Using OpenMP*".
- You can use environment variables to control the behavior of the PGDBG debugger or PGPROF profiler. For a description of environment variables that affect these tools, refer to the PGI Tools Guide.

Setting Environment Variables

Before we look at the environment variables that you might use with the PGI compilers and tools, let's take a look at how to set environment variables. To illustrate how to set these variables in various environments, lets look at how a user might initialize the shell environment prior to using the PGI compilers and tools.

Setting Environment Variables on Linux

Let's assume that you want access to the PGI products when you log on. Let's further assume that you installed the PGI compilers in `/opt/pgi` and that the license file is in `/opt/pgi/license.dat`. For access at startup, you can add the following lines to your startup file.

In csh, use these commands:

```
% setenv PGI /opt/pgi
% setenv MANPATH "$MANPATH":$PGI/linux86/9.0/man
% setenv LM_LICENSE_FILE $PGI/license.dat
% set path = ($PGI/linux86/9.0/bin $path)
```

In bash, sh, zsh, or ksh, use these commands:

```
$ PGI=/opt/pgi; export PGI
```

```
$ MANPATH=$MANPATH:$PGI/linux86/9.0/man; export MANPATH
$ LM_LICENSE_FILE=$PGI/license.dat; export LM_LICENSE_FILE
$ PATH=$PGI/linux86/9.0/bin:$PATH; export PATH
```

Setting Environment Variables on Windows

In Windows, when you access PGI Workstation 9.0 (*Start* | *PGI Workstation 9.0*), you have two options that PGI provides for setting your environment variables - either the DOS command environment or the Cygwin Bash environment. When you open either of these shells available to you, the default environment variables are already set and available to you.

You may want to use other environment variables, such as the OpenMP ones. This section explains how to do that.

Suppose that your home directory is `C:\tmp`. The following examples show how you might set the temporary directory to your home directory, and then verify that it is set.

Command prompt:

From PGI Workstation 9.0, select *PGI Workstation Tools* | *PGI Command Prompt* (32-bit or 64-bit), and enter the following:

```
DOS> set TMPDIR=C:\tmp
DOS> echo %TMPDIR%
C:\tmp
DOS>
```

Cygwin Bash prompt:

From PGI Workstation 9.0, select PGI Workstation (32-bit or 64-bit) and at the Cygwin Bash prompt, enter the following

```
PGI$ export TMPDIR=C:\tmp
PGI$ echo $TMPDIR
C:\tmp
PGI$
```

Setting Environment Variables on Mac OSX

Let's assume that you want access to the PGI products when you log on. Let's further assume that you installed the PGI compilers in `/opt/pgi` and that the license file is in `/opt/pgi/license.dat`. For access at startup, you can add the following lines to your startup file.

For x64 osx86-64 in a csh:

```
% set path = (/opt/pgi/osx86-64/9.0/bin $path)
% setenv MANPATH "$MANPATH":/opt/pgi/osx86-64/9.0/man
```

For x64 osx86-64 in a bash, sh, zsh, or ksh:

```
$ PATH=/opt/pgi/osx86-64/9.0/bin:$PATH; export PATH
$ MANPATH=$MANPATH:/opt/pgi/osx86-64/9.0/man; export MANPATH
```

PGI-Related Environment Variables

For easy reference, the following table provides a quick listing of some OpenMP and all PGI compiler-related environment variables. This section provides more detailed descriptions of the environment variables specific to PGI compilers and the executables they generate. For information specific to OpenMP environment variables, refer to Table 5.4, "OpenMP-related Environment Variable Summary Table," on page 71 and to the complete descriptions in "OpenMP Environment Variables"

Table 10.1. PGI-Related Environment Variable Summary

Environment Variable	Description
FLEXLM_BATCH	(Windows only) When set to 1, prevents interactive pop-ups from appearing by sending all licensing errors and warnings to standard out rather than to a pop-up window.
FORTRANOPT	Allows the user to specify that the PGI Fortran compilers user VAX I/O conventions.
GMON_OUT_PREFIX	Specifies the name of the output file for programs that are compiled and linked with the –pg option.
LD_LIBRARY_PATH	Specifies a colon-separated set of directories where libraries should first be searched, prior to searching the standard set of directories.
LM_LICENSE_FILE	Specifies the full path of the license file that is required for running the PGI software. On Windows, `LM_LICENSE_FILE` does not need to be set.
MANPATH	Sets the directories that are searched for manual pages associated with the command that the user types.
MPSTKZ	Increases the size of the stacks used by threads executing in parallel regions. The value should be an integer <n> concatenated with `M` or `m` to specify stack sizes of n megabytes.
MP_BIND	Specifies whether to bind processes or threads executing in a parallel region to a physical processor.
MP_BLIST	When `MP_BIND` is `yes`, this variable specifically defines the thread-CPU relationship, overriding the default values.
MP_SPIN	Specifies the number of times to check a semaphore before calling sched_yield() (on Linux, SUA, or Mac OS X) or _sleep() (on Windows).
MP_WARN	Allows you to eliminate certain default warning messages.
NCPUS	Sets the number of processes or threads used in parallel regions.
NCPUS_MAX	Limits the maximum number of processors or threads that can be used in a parallel region.
NO_STOP_MESSAGE	If used, the execution of a plain STOP statement does not produce the message FORTRAN STOP.

Environment Variable	Description
OMP_DYNAMIC	Currently has no effect. Enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads. The default is FALSE.
OMP_MAX_ACTIVE_LEVELS	Currently has no effect. Enables (TRUE) or disables (FALSE) nested parallelism. The default is FALSE.
OMP_NUM_THREADS	Specifies the number of threads to use during execution of parallel regions. Default is 1.
OMP_SCHEDULE	Specifies the type of iteration scheduling and, optionally, the chunk size to use for *omp for* and *omp parallel for* loops that include the run-time schedule clause. The default is STATIC with chunk size = 1.
OMP_STACKSIZE	Overrides the default stack size for a newly created thread.
OMP_WAIT_POLICY	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE. The default is ACTIVE.
PATH	Determines which locations are searched for commands the user may type.
PGI	Specifies, at compile-time, the root directory where the PGI compilers and tools are installed.
PGI_CONTINUE	If set, when a program compiled with -Mchkfpstk is executed, the stack is automatically cleaned up and execution then continues.
PGI_OBJSUFFIX	Allows you to control the suffix on generated object files.
PGI_STACK_USAGE	(Windows only) Allows you to explicitly set stack properties for your program.
PGI_TERM	Controls the stack traceback and just-in-time debugging functionality.
PGI_TERM_DEBUG	Overrides the default behavior when PGI_TERM is set to debug.
PWD	Allows you to display the current directory.
STATIC_RANDOM_SEED	Forces the seed returned by RANDOM_SEED to be constant.
TMP	Sets the directory to use for temporary files created during execution of the PGI compilers and tools; interchangeable with TMPDIR.
TMPDIR	Sets the directory to use for temporary files created during execution of the PGI compilers and tools.

PGI Environment Variables

You use the environment variables listed in Table 10.1 to alter the default behavior of the PGI compilers and the executables which they generate. This section provides more detailed descriptions about the variables in this table that are not OpenMP environment variables.

FLEXLM_BATCH

By default, on Windows the license server creates interactive pop-up messages to issue warning and errors. You can use the environment variable FLEXLM_BATCH to prevent interactive pop-up windows. To do this, set the environment variable FLEXLM_BATCH to 1.

The following csh example prevents interactive pop-up messages for licensing warnings and errors:

```
% set FLEXLM_BATCH = 1;
```

FORTRANOPT

FORTRANOPT allows the user to adjust the behavior of the PGI Fortran compilers.

- If FORTRANOPT exists and contains the value vaxio, the record length in the open statement is in units of 4-byte words, and the $ edit descriptor only has an effect for lines beginning with a space or a plus sign (+).
- If FORTRANOPT exists and contains the value format_relaxed, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) is not required to be a type implied by the descriptor.
- In a non-Windows environment, if FORTRANOPT exists and contains the value crif, a sequential formatted or list-directed record is allowed to be terminated with the character sequence \r\n (carriage return, newline). This approach is useful when reading records from a file produced on a Window's system.

The following example causes the PGI Fortran compilers to use VAX I/O conventions:

```
% setenv FORTRANOPT vaxio
```

GMON_OUT_PREFIX

GMON_OUT_PREFIX specifies the name of the output file for programs that are compiled and linked with the -pg option. The default name is gmon.out.a.

If GMON_OUT_PREFIX is set, the name of the output file has GMON_OUT_PREFIX as a prefix. Further, the suffix is the pid of the running process. The prefix and suffix are separated by a dot. For example, if the output file is mygmon, then the full filename may look something similar to this: GMON_OUT_PREFIX.mygmon.0012348567.

The following example causes the PGI Fortran compilers to use pgout as the output file for programs compiled and linked with the -pg option.

```
% setenv GMON_OUT_PREFIX pgout
```

LD_LIBRARY_PATH

The LD_LIBRARY_PATH variable is a colon-separated set of directories specifying where libraries should first be searched, prior to searching the standard set of directories. This variable is useful when debugging a new library or using a nonstandard library for special purposes.

The following csh example adds the current directory to your LD_LIBRARY_PATH variable.

```
% setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH":"./"
```

LM_LICENSE_FILE

The LM_LICENSE_FILE variable specifies the full path of the license file that is required for running the PGI software.

For example, once the license file is in place, you can execute the following csh commands to make the products you have purchased accessible and to initialize your environment for use of FLEXlm. These commands assume that you use the default installation directory: /opt/pgi

```
% setenv PGI /opt/pgi
% setenv LM_LICENSE_FILE "$LM_LICENSE_FILE":/opt/pgi/license.dat
```

To set the environment variable LM_LICENSE_FILE to the full path of the license key file, do this:

1. Open the System Properties dialog: *Start* | *Control Panel* | *System*.
2. Select the *Advanced* tab.
3. Click the *Environment Variables* button.
   - If LM_LICENSE_FILE is not already an environment variable, create a new system variable for it. Set its value to the full path, including the name of the license key file, license.dat.
   - If LM_LICENSE_FILE already exists as an environment variable, append the path to the license file to the variable's current value using a semi-colon to separate entries.

MANPATH

The MANPATH variable sets the directories that are searched for manual pages associated with the commands that the user types. When using PGI products, it is important that you set your PATH to include the location of the PGI products and then set the MANPATH variable to include the man pages associated with the products.

The following csh example targets x64 linux86-64 version of the compilers and tool s and allows the user access to the manual pages associated with them.

```
% set path = (/opt/pgi/linux86-64/9.0/bin $path
% setenv MANPATH "$MANPATH":/opt/pgi/linux86-64/9.0/man
```

MPSTKZ

MPSTKZ increases the size of the stacks used by threads executing in parallel regions. You typically use this variable with programs that utilize large amounts of thread-local storage in the form of private variables or local variables in functions or subroutines called within parallel regions. The value should be an integer <n> concatenated with M or m to specify stack sizes of n megabytes.

For example, the following setting specifies a stack size of 8 megabytes.

```
% setenv MPSTKZ 8M
```

MP_BIND

You can set MP_BIND to yes or y to bind processes or threads executing in a parallel region to physical processor. Set it to no or n to disable such binding. The default is to not bind processes to processors. This variable is an execution-time environment variable interpreted by the PGI run-time support libraries. It does not affect the behavior of the PGI compilers in any way.

Note

The MP_BIND environment variable is not supported on all platforms.

```
% setenv MP_BIND y
```

MP_BLIST

MP_BLIST allows you to specifically define the thread-CPU relationship.

Note

This variable is only in effect when MP_BIND is yes.

While the MP_BIND variable binds processors or threads to a physical processor, MP_BLIST allows you to specifically define which thread is associated with which processor. The list defines the processor-thread relationship order, beginning with thread 0. This list overrides the default binding.

For example, the following setting for MP_BLIST maps CPUs 3, 2, 1 and 0 to threads 0, 1, 2 and 3 respectively.

```
% setenv MP_BLIST=3,2,1,0
```

MP_SPIN

When a thread executing in a parallel region enters a barrier, it spins on a semaphore. You can use MP_SPIN to specify the number of times it checks the semaphore before calling sched_yield() (on Linux, SUA, or MAC OS X) or _sleep() (on Windows). These calls cause the thread to be re-scheduled, allowing other processes to run. The default value is 1000000..

```
% setenv MP_SPIN 200
```

MP_WARN

MP_WARN allows you to eliminate certain default warning messages.

By default, a warning is printed to stderr if you execute an OpenMP or auto-parallelized program with NCPUS or OMP_NUM_THREADS set to a value larger than the number of physical processors in the system.

For example, if you produce a parallelized executable a.out and execute as follows on a system with only one processor, you get a warning message.

```
% setenv OMP_NUM_THREADS 2
% a.out
Warning: OMP_NUM_THREADS or NCPUS (2) greater
than available cpus (1)
FORTRAN STOP
```

Setting MP_WARN to NO eliminates these warning messages.

NCPUS

You can use the NCPUS environment variable to set the number of processes or threads used in parallel regions. The default is to use only one process or thread, which is known as serial mode.

Note

OMP_NUM_THREADS has the same functionality as NCPUS. For historical reasons, PGI supports the environment variable NCPUS. If both OMP_NUM_THREADS and NCPUS are set, the value of OMP_NUM_THREADS takes precedence.

Warning

Setting NCPUS to a value larger than the number of physical processors or cores in your system can cause parallel programs to run very slowly.

NCPUS_MAX

You can use the NCPUS_MAX environment variable to limit the maximum number of processes or threads used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using set_omp_num_threads(), will cause the number of processes or threads to be set at the value of NCPUS_MAX rather than the value specified in the function call.

NO_STOP_MESSAGE

If the NO_STOP_MESSAGE variable exists, the execution of a plain STOP statement does not produce the message FORTRAN STOP. The default behavior of the PGI Fortran compilers is to issue this message.

PATH

The PATH variable sets the directories that are searched for commands that the user types. When using PGI products, it is important that you set your PATH to include the location of the PGI products.

You can also use this variable to specify that you want to use only the linux86 version of the compilers and tools, or to target linux86 as the default.

The following csh example targets x64 linux86-64 version of the compilers and tools.

```
% set path = (/opt/pgi/linux86-64/9.0/bin $path)
```

PGI

The PGI environment variable specifies the root directory where the PGI compilers and tools are installed. This variable is recognized at compile-time. If it is not set, the default value depends on your system as well as which compilers are installed:

- On Linux, the default value of this variable is /opt/pgi.
- On Windows, the default value is C:\Program Files\PGI, where C represents the system drive. If both 32- and 64-bit compilers are installed, the 32-bit compilers are in C:\Program Files (x86)\PGI .
- On SFU/SUA and Mac OS X, the default value of this variable is /opt/pgi. The corresponding Windows-style path is C:\SFU\opt\pgi for SFU and C:\WINDOWS\SUA\opt\pgi for SUA, where C represents the system drive.

In most cases, if the PGI environment variable is not set, the PGI compilers and tools dynamically determine the location of this root directory based on the instance of the compiler or tool that was invoked. However,

there are still some dependencies on the PGI environment variable, and you can use it as a convenience when initializing your environment for use of the PGI compilers and tools.

For example, assuming you use csh and want the 64-bit linux86-64 versions of the PGI compilers and tools to be the default, you would use this syntax:

```
% setenv PGI /usr/pgi
% setenv MANPATH "$MANPATH":$PGI/linux86/9.0/man
% setenv LM_LICENSE_FILE $PGI/license.dat
% set path = ($PGI/linux86-64/9.0/bin $path)
```

PGI_CONTINUE

You set the `PGI_CONTINUE` variable to specify the actions to take before continuing with execution. For example, if the `PGI_CONTINUE` environment variable is set and then a program that is compiled with `-Mchkfpstk` is executed, the stack is automatically cleaned up and execution then continues. If `PGI_CONTINUE` is set to `verbose`, the stack is automatically cleaned up, a warning message is printed, and then execution continues.

Note

There is a performance penalty associated with the stack cleanup.

PGI_OBJSUFFIX

You can set the `PGI_OBJSUFFIX` environment variable to generate object files that have a specific suffix. For example, if you set `PGI_OBJSUFFIX` to `.o`, the object files have a suffix of `.o` rather than `.obj`.

PGI_STACK_USAGE

(Windows only) The `PGI_STACK_USAGE` variable allows you to explicitly set stack properties for your program. When the user compiles a program with the `-Mchkstk` option and sets the PGI_STACK_USAGE environment variable to any value, the program displays the stack space allocated and used after the program exits. You might see something similar to the following message:

```
thread 0 stack: max 8180KB, used 48KB
```

This message indicates that the program used 48KB of a 8180KB allocated stack. For more information on the `-Mchkstk` option, refer to –Mchkstk.

PGI_TERM

The `PGI_TERM` environment variable controls the stack traceback and just-in-time debugging functionality. The runtime libraries use the value of `PGI_TERM` to determine what action to take when a program abnormally terminates.

The value of `PGI_TERM` is a comma-separated list of options. The commands for setting the environment variable follow.

- In csh:

```
% setenv PGI_TERM option[,option...]
```

- In bash, sh, zsh, or ksh:

```
$ PGI_TERM=option[,option...]
$ export PGI_TERM
```

- In the Windows Command Prompt:

```
C:\> set PGI_TERM=option[,option...]
```

Table 10.2 lists the supported values for `option`. Following the table is a complete description of each option that indicates specifically how you might apply the option.

By default, all of these options are disabled.

Table 10.2. Supported PGI_TERM Values

[no]debug	Enables/disables just-in-time debugging (debugging invoked on error)
[no]trace	Enables/disables stack traceback on error
[no]signal	Enables/disables establishment of signal handlers for common signals that cause program termination
[no]abort	Enables/disables calling the system termination routine abort()

[no]debug

This enables/disables just-in-time debugging. The default is `nodebug`.

When `PGI_TERM` is set to `debug`, the following command is invoked on error, unless you use `PGI_TERM_DEBUG` to override this default.

```
pgdbg -text -attach <pid>
```

`<pid>` is the process ID of the process being debugged.

The PGI_TERM_DEBUG environment variable may be set to override the default setting. For more information, refer to "PGI_TERM_DEBUG," on page 137.

[no]trace

This enables/disables the stack traceback. The default is `notrace`.

[no]signal

This enables/disables establishing signal handlers for the most common signals that cause program termination. The default is `nosignal`. Setting `trace` and `debug` automatically enables `signal`. Specifically setting `nosignal` allows you to override this behavior.

[no]abort

This enables/disables calling the system termination routine abort(). The default is `noabort`. When `noabort` is in effect the process terminates by calling `_exit(127)`.

On Linux and SUA, when `abort` is in effect, the abort routine creates a core file and exits with code 127.

On Windows, when `abort` is in effect, the abort routine exits with the status of the exception received. For example, if the program receives an access violation, abort() exits with status 0xC0000005.

A few runtime errors just print an error message and call `exit(127)`, regardless of the status of `PGI_TERM`. These are mainly errors such as specifying an invalid environment variable value where a traceback would not be useful.

If it appears that abort() does not generate core files on a Linux system, be sure to unlimit the coredumpsize. You can do this in these ways:

- Using csh:

```
% limit coredumpsize unlimited
% setenv PGI_TERM abort
```

- Using bash, sh, zsh, or ksh:

```
$ ulimit -c unlimited
$ export PGI_TERM=abort
```

To debug a core file with pgdbg, start pgdbg with the -core option. For example, to view a core file named "core" for a program named "a.out":

```
$ pgdbg -core core a.out
```

For more information on why to use this variable, refer to "Stack Traceback and JIT Debugging," on page 139.

PGI_TERM_DEBUG

The `PGI_TERM_DEBUG` variable may be set to override the default behavior when `PGI_TERM` is set to `debug`.

The value of `PGI_TERM_DEBUG` should be set to the command line used to invoke the program. For example:

```
gdb --quiet --pid %d
```

The first occurrence of `%d` in the `PGI_TERM_DEBUG` string is replaced by the process id. The program named in the `PGI_TERM_DEBUG` string must be found on the current `PATH` or specified with a full path name.

PWD

The PWD variable allows you to display the current directory.

STATIC_RANDOM_SEED

You can use `STATIC_RANDOM_SEED` to force the seed returned by the Fortran 90/95 `RANDOM_SEED` intrinsic to be constant. The first call to `RANDOM_SEED` without arguments resets the random seed to a default value, then advances the seed by a variable amount based on time. Subsequent calls to `RANDOM_SEED` without arguments reset the random seed to the same initial value as the first call. Unless the time is exactly the same, each time a program is run a different random number sequence is generated. Setting the environment variable `STATIC_RANDOM_SEED` to `YES` forces the seed returned by `RANDOM_SEED` to be constant, thereby generating the same sequence of random numbers at each execution of the program.

TMP

You can use TMP to specify the directory to use for placement of any temporary files created during execution of the PGI compilers and tools. This variable is interchangeable with TMPDIR.

TMPDIR

You can use TMPDIR to specify the directory to use for placement of any temporary files created during execution of the PGI compilers and tools.

Using Environment Modules on Linux

On Linux, if you use the Environment Modules package, that is, the module load command, PGI includes a script to set up the appropriate module files.

Assuming your installation base directory is /opt/pgi, and your MODULEPATH environment variable is /usr/local/Modules/modulefiles, execute this command:

```
% /opt/pgi/linux86/9.0-0/etc/modulefiles/pgi.module.install \
 -all -install /usr/local/Modules/modulefiles
```

This command creates module files for all installed versions of the PGI compilers. You must have write permission to the modulefiles directory to enable the module commands:

```
% module load pgi32/9.0
% module load pgi64/9.0
% module load pgi/9.0
```

where "pgi/9.0" uses the 32-bit compilers on a 32-bit system and uses 64-bit compilers on a 64-bit system.

To see what versions are available, use this command:

```
% module avail pgi
```

The module load command sets or modifies the environment variables as indicated in the following table.

This Environment Variable...	Is set or modified by the module load command to ...
CC	Full path to pgcc
CPP	Full path to pgCC
CXX	Path to pgCC
C++	Path to pgCC
FC	Full path to pgf95
F77	Full path to pgf77
F90	Full path to pgf90
LD_LIBRARY_PATH	Prepends the PGI library directory
MANPATH	Prepends the PGI man page directory
PATH	Prepends the PGI compiler and tools bin directory

This Environment Variable...	Is set or modified by the module load command to ...
PGI	The base installation directory
V	Full path to pgCC

Note

PGI does not provide support for the Environment Modules package. For more information about the package, go to: http://modules.sourceforge.net.

Stack Traceback and JIT Debugging

When a programming error results in a run-time error message or an application exception, a program will usually exit, perhaps with an error message. The PGI run-time library includes a mechanism to override this default action and instead print a stack traceback, start a debugger, or, on Linux, create a core file for post-mortem debugging.

The stack traceback and just-in-time debugging functionality is controlled by an environment variable, PGI_TERM, described in "PGI_TERM," on page 135. The run-time libraries use the value of PGI_TERM to determine what action to take when a program abnormally terminates.

When the PGI runtime library detects an error or catches a signal, it calls the routine pgi_stop_here() prior to generating a stack traceback or starting the debugger. The pgi_stop_here() routine is a convenient spot to set a breakpoint when debugging a program.

Chapter 11. Distributing Files - Deployment

Once you have successfully built, debugged and tuned your application, you may want to distribute it to users who need to run it on a variety of systems. This chapter addresses how to effectively distribute applications built using PGI compilers and tools. The application must be installed in such a way that it executes accurately on a system other than the one on which it was built, and which may be configured differently.

Deploying Applications on Linux

To successfully deploy your application on Linux, there are a number of issues to consider, including these:

- Runtime Libraries
- 64-bit Linux Systems
- Redistribution of Files
- Linux Portability of files and packages
- Licensing

Runtime Library Considerations

On Linux systems, the system runtime libraries can be linked to an application either statically, or dynamically, For example, for the C runtime library, libc, you can use either the static version `libc.a` or the shared object `libc.so`. If the application is intended to run on Linux systems other than the one on which it was built, it is generally safer to use the shared object version of the library. This approach ensures that the application uses a version of the library that is compatible with the system on which the application is running. Further, it works best when the application is linked on a system that has an equivalent or earlier version of the system software than the system on which the application will be run.

Note

Building on a newer system and running the application on an older system may not produce the desired output.

To use the shared object version of a library, the application must also link to shared object versions of the PGI runtime libraries. To execute an application built in such a way on a system on which PGI compilers are *not* installed, those shared objects must be available.To build using the shared object versions of the runtime libraries, use the `-Bdynamic` option, as shown here:

```
$ pgf90 -Bdynamic myprog.f90
```

64-bit Linux Considerations

On 64-bit Linux systems, 64-bit applications that use the `-mcmodel=medium` option sometimes cannot be successfully linked statically. Therefore, users with executables built with the `-mcmodel=medium` option may need to use shared libraries, linking dynamically. Also, runtime libraries built using the `-fpic` option use 32-bit offsets, so they sometimes need to reside near other runtime `libs` in a shared area of Linux program memory.

Note

If your application is linked dynamically using shared objects, then the shared object versions of the PGI runtime are required.

Linux Redistributable Files

There are two methods for installing the shared object versions of the runtime libraries required for applications built with PGI compilers and tools: Linux Portability Package and manual distribution.

PGI provides the Linux Portability Package, an installation package that can be downloaded from the PGI web site. In addition, when the PGI compilers are installed, there is a directory named `REDIST` for each platform (linux86 and linux86-64) that contains the redistributed shared object libraries. These may be redistributed by licensed PGI customers under the terms of the PGI End-User License Agreement.

Restrictions on Linux Portability

You cannot expect to be able to run an executable on any given Linux machine. Portability depends on the system you build on as well as how much your program uses system routines that may have changed from Linux release to Linux release. For example, one area of significant change between some versions of Linux is in `libpthread.so`. PGI compilers use this shared object for both the option `-Mconcur` (auto-parallel) and the option `-mp` (OpenMP) programs.

Typically, portability is supported for forward execution, meaning running a program on the same or a later version of Linux; but not for backward compatibility, that is, running on a prior release. For example, a user who compiles and links a program under Suse 9.1 should not expect the program to run without incident on a Red Hat 9.0 system, which is an earlier version of Linux. It *may* run, but it is less likely. Developers might consider building applications on earlier Linux versions for wider usage.

Installing the Linux Portability Package

You can download the Linux Portability Packages from the Downloads page at http://www.pgroup.com. First download the package you need, then untar it, and run the install script. Then you can add the installation directory to your library path.

To use the installed libraries, you can either modify `/etc/ld.so.conf` and run `ldconfig(1)` or modify the environment variable `LD_LIBRARY_PATH`, as shown here:

```
setenv LD_LIBRARY_PATH /usr/local/pgi
```

or

```
export LD_LIBRARY_PATH=/usr/local/pgi
```

Licensing for Redistributable Files

The installation of the Linux Portability Package presents the standard PGI usage license. The `libs` can be distributed for use with PGI compiled applications, within the provisions of that license.

The files in the REDIST directories may be redistributed under the terms of the End-User License Agreement for the product in which they were included.

Deploying Applications on Windows

Windows programs may be linked statically or dynamically.

- A statically linked program is completely self-contained, created by linking to static versions of the PGI and Microsoft runtime libraries.
- A dynamically linked program depends on separate dynamically-linked libraries (DLLs) that must be installed on a system for the application to run on that system.

Although it may be simpler to install a statically linked executable, there are advantages to using the DLL versions of the runtime, including these:

- Executable binary file size is smaller.
- Multiple processes can use DLLs at once, saving system resources.
- New versions of the runtime can be installed and used by the application without rebuilding the application.

Dynamically-linked Windows programs built with PGI compilers depend on dynamic run-time library files (DLLs). These DLLs must be distributed with such programs to enable them to execute on systems where the PGI compilers are not installed. These redistributable libraries include both PGI runtime libraries and Microsoft runtime libraries.

PGI Redistributables

PGI redistributable directories contain all of the PGI Linux runtime library shared object files or Windows dynamically-linked libraries that can be re-distributed by PGI 9.0 licensees under the terms of the PGI End-user License Agreement (EULA).

Microsoft Redistributables

The PGI products on Windows include Microsoft Open Tools. The Microsoft Open Tools directory contains a subdirectory named `redist`. PGI licensees may redistribute the files contained in this directory in accordance with the terms of the PGI End-User License Agreement.

Microsoft supplies installation packages, `vcredist_x86.exe` and `vcredist_x64.exe`, containing these runtime files. These files are available in the redist directory.

Code Generation and Processor Architecture

The PGI compilers can generate much more efficient code if they know the specific x86 processor architecture on which the program will run. When preparing to deploy your application, you should determine whether you want the application to run on the widest possible set of x86 processors, or if you want to restrict the application to run on a specific processor or set of processors. The restricted approach allows you to optimize performance for that set of processors.

Different processors have differences, some subtle, in hardware features, such as instruction sets and cache size. The compilers make architecture-specific decisions about such things as instruction selection, instruction scheduling, and vectorization, all of which can have a profound effect on the performance of your application.

Processor-specific code generation is controlled by the `-tp` option, described in "–tp <target> [,target...],"
on page 236. When an application is compiled without any `-tp` options, the compiler generates code for the type of processor on which the compiler is run.

Generating Generic x86 Code

To generate generic x86 code, use one of the following forms of the `-tp` option on your command line:

```
-tp px ! generate code for any x86 cpu type
```

```
-tp p6 ! generate code for Pentium 2 or greater
```

While both of these examples are good choices for portable execution, most users have Pentium 2 or greater CPUs.

Generating Code for a Specific Processor

You can use the `-tp` option to request that the compiler generate code optimized for a specific processor. The PGI Release Notes contains a list of supported processors or you can look at the `-tp` entry in the compiler output generated by using the `-help` option, described in "–help ," on page 212.

Generating One Executable for Multiple Types of Processors

PGI unified binaries provide a low-overhead method for a single program to run well on a number of hardware platforms.

All 64-bit PGI compilers can produce PGI Unified Binary programs that contain code streams fully optimized and supported for both AMD64 and Intel EM64T processors using the `-tp` target option.

The compilers generate and combine multiple binary code streams into one executable, where each stream is optimized for a specific platform. At runtime, this one executable senses the environment and dynamically selects the appropriate code stream.

Executable size is automatically controlled via unified binary culling. Only those functions and subroutines where the target affects the generated code have unique binary images, resulting in a code-size savings of 10-90% compared to generating full copies of code for each target.

Programs can use PGI Unified Binary technology even if all of the object files and libraries are not compiled as unified binaries. Like any other object file, you can use PGI Unified Binary object files to create programs or libraries. No special start up code is needed; support is linked in from the PGI libraries.

The `-Mpfi` option disables generation of PGI Unified Binary object files. Instead, the default target auto-detect rules for the host are used to select the target processor.

PGI Unified Binary Command-line Switches

The PGI Unified Binary command-line switch is an extension of the target processor switch, `-tp`, which may be applied to individual files during compilation .

The target processor switch, `-tp`, accepts a comma-separated list of 64-bit targets and generates code optimized for each listed target.

The following example generates optimized code for three targets:

```
-tp k8-64,p7-64,core2-64
```

A special target switch, `-tp x64`, is the same as `-tp k8-64, p7-64s`.

PGI Unified Binary Directives and Pragmas

PGI Unified binary directives and pragmas may be applied to functions, subroutines, or whole files. The directives and pragmas cause the compiler to generate PGI Unified Binary code optimized for one or more targets. No special command line options are needed for these pragmas and directives to take effect.

The syntax of the Fortran directive is this:

```
pgi$[g|r| ] pgi tp [target]...
```

where the scope is g (global), r (routine) or blank. The default is r, routine.

For example, the following syntax indicates that the whole file, represented by g, should be optimized for both k8_64 and p7_64.

```
pgi$g pgi tp k8_64 p7_64
```

The syntax of the C/C++ pragma is this:

```
#pragma [global|routine|] tp [target]...
```

where the scope is global, routine, or blank. The default is routine.

For example, the following syntax indicates that the next function should be optimized for k8_64, p7_64, and core2_64.

```
#pragma routine tp k8_64 p7_64 core2_64
```

Chapter 12. Inter-language Calling

This chapter describes inter-language calling conventions for C, C++, and Fortran programs using the PGI compilers. The following sections describe how to call a Fortran function or subroutine from a C or C++ program and how to call a C or C++ function from a Fortran program. For information on calling assembly language programs, refer to Chapter 21, "*Run-time Environment*".

This chapter provides examples that use the following options related to inter-language calling. For more information on these options, refer to Chapter 16, "*Command-Line Options Reference*," on page 195.

`-c` `-Mnomain` `-Miface` `-Mupcase`

Overview of Calling Conventions

This chapter includes information on the following topics:

- Functions and subroutines in Fortran, C, and C++
- Naming and case conversion conventions
- Compatible data types
- Argument passing and special return values
- Arrays and indexes
- Win32 calling conventions

The sections "Inter-language Calling Considerations," on page 148 through"Example - C++ Calling Fortran," on page 157 describe how to perform inter-language calling using the Linux, Mac OSX, Win64 or SUA convention. Default Fortran calling conventions for Win32 differ, although Win32 programs compiled using the `-Miface=unix` Fortran command-line option use the Linux/Win64 convention rather than the default Win32 conventions. All information in those sections pertaining to compatibility of arguments applies to Win32 as well. For details on the symbol name and argument passing conventions used on Win32 platforms, refer to "Win32 Calling Conventions," on page 158.

Inter-language Calling Considerations

In general, when argument data types and function return values agree, you can call a C or C++ function from Fortran as well as call a Fortran function from C or C++. When data types for arguments do not agree, you may need to develop custom mechanisms to handle them. For example, the Fortran `COMPLEX` type has a matching type in C99 but does not have a matching type in C90; however, it is still possible to provide inter-language calls but there are no general calling conventions for such cases.

Note

- If a C++ function contains objects with constructors and destructors, calling such a function from either C or Fortran is not possible unless the initialization in the main program is performed from a C++ program in which constructors and destructors are properly initialized.
- In general, you can call a C or Fortran function from C++ without problems as long as you use the extern "C" keyword to declare the function in the C++ program. This declaration prevents name mangling for the C function name. If you want to call a C++ function from C or Fortran, you also have to use the extern "C" keyword to declare the C++ function. This keeps the C++ compiler from mangling the name of the function.
- You can use the __cplusplus macro to allow a program or header file to work for both C and C++. For example, the following defines in the header file stdio.h allow this file to work for both C and C++.

```
#ifndef _STDIO_H
#define _STDIO_H
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
. /* Functions and data types defined... */
.
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif
```

- C++ member functions cannot be declared `extern`, since their names will always be mangled. Therefore, C++ member functions cannot be called from C or Fortran.

Functions and Subroutines

Fortran, C, and C++ define functions and subroutines differently.

For a Fortran program calling a C or C++ function, observe the following return value convention:

- When a C or C++ function returns a value, call it from Fortran as a function.
- When a C or C++ function does not return a value, call it as a subroutine.

For a C/C++ program calling a Fortran function, the call should return a similar type. Table 12.1, “Fortran and C/C++ Data Type Compatibility,” on page 149 lists compatible types. If the call is to a Fortran subroutine,

a Fortran CHARACTER function, or a Fortran COMPLEX function, call it from C/C++ as a function that returns void. The exception to this convention is when a Fortran subroutine has alternate returns; call such a subroutine from C/C++ as a function returning int whose value is the value of the integer expression specified in the alternate RETURN statement.

Upper and Lower Case Conventions, Underscores

By default on Linux, Win64, OSX, and SUA systems, all Fortran symbol names are converted to lower case. C and C++ are case sensitive, so upper-case function names stay upper-case. When you use inter-language calling, you can either name your C/C++ functions with lower-case names, or invoke the Fortran compiler command with the option -Mupcase, in which case it will not convert symbol names to lower-case.

When programs are compiled using one of the PGI Fortran compilers on Linux, Win64, OSX, and SUA systems, an underscore is appended to Fortran global names (names of functions, subroutines and common blocks). This mechanism distinguishes Fortran name space from C/C++ name space. Use these naming conventions:

- If you call a C/C++ function from Fortran, you should rename the C/C++ function by appending an underscore or use C$PRAGMA C in the Fortran program. For more information on C$PRAGMA C, refer to "C$PRAGMA C," on page 110.
- If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

Compatible Data Types

Table 12.1 shows compatible data types between Fortran and C/C++. Table 12.2, "Fortran and C/C++ Representation of the COMPLEX Type," on page 150 shows how the Fortran COMPLEX type may be represented in C/C++.

Tip

If you can make your function/subroutine parameters as well as your return values match types, you should be able to use inter-language calling.

Table 12.1. Fortran and C/C++ Data Type Compatibility

Fortran Type (lower case)	C/C++ Type	Size (bytes)
character x	char x	1
character*n x	char x[n]	n
real x	float x	4
real*4 x	float x	4
real*8 x	double x	8
double precision	double x	8
integer x	int x	4
integer*1 x	signed char x	1

Fortran Type (lower case)	C/C++ Type	Size (bytes)
integer*2 x	short x	2
integer*4 x	int x	4
integer*8 x	long long x	8
logical x	int x	4
logical*1 x	char x	1
logical*2 x	short x	2
logical*4	int x	4
logical*8	long long x	8

Table 12.2. Fortran and C/C++ Representation of the **COMPLEX** Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x;	8
	float complex x;	8
complex*8 x	struct {float r,i;} x;	8
	float complex x;	8
double complex x	struct {double dr,di;} x;	16
	double complex x;	16
complex *16 x	struct {double dr,di;} x;	16
	double complex x;	16

Note

For C/C++, the `complex` type implies C99 or later.

Fortran Named Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore. For example, here is a Fortran common block:

```
INTEGER I
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, c, cd, d
```

This Fortran Common Block is represented in C with the following equivalent:

```
extern struct {
 int i;
 struct {float real, imag;} c;
```

```
 struct {double real, imag;} cd;
 double d;
} com_;
```

This same Fortran Common Block is represented in C++ with the following equivalent:

```
extern "C" struct {
 int i;
 struct {float real, imag;} c;
 struct {double real, imag;} cd;
 double d;
} com_;
```

Tip

For global or external data sharing, `extern "C"` is not required.

Argument Passing and Return Values

In Fortran, arguments are passed by reference, that is, the address of the argument is passed, rather than the argument itself. In C/C++, arguments are passed by value, except for strings and arrays, which are passed by reference. Due to the flexibility provided in C/C++, you can work around these differences. Solving the parameter passing differences generally involves intelligent use of the `&` and `*` operators in argument passing when C/C++ calls Fortran and in argument declarations when Fortran calls C/C++.

For strings declared in Fortran as type `CHARACTER`, an argument representing the length of the string is also passed to a calling function.

On Linux, SUA, and Mac OS X systems, or when using the UNIX calling convention on Windows (option `-Miface=unix`), the compiler places the length argument(s) at the end of the parameter list, following the other formal arguments. The length argument is passed by value, not by reference.

Passing by Value (%VAL)

When passing parameters from a Fortran subprogram to a C/C++ function, it is possible to pass by value using the `%VAL` function. If you enclose a Fortran parameter with `%VAL()`, the parameter is passed by value. For example, the following call passes the integer `i` and the logical `bvar` by value.

```
integer*1 i
logical*1 bvar
call cvalue (%VAL(i), %VAL(bvar))
```

Character Return Values

"Functions and Subroutines," on page 148 describes the general rules for return values for C/C++ and Fortran inter-language calling. There is a special return value to consider. When a Fortran function returns a character, two arguments need to be added at the beginning of the C/C++ calling function's argument list:

- The address of the return character or characters
- The length of the return character

Example 12.1, "Character Return Parameters" illustrates the extra parameters, `tmp` and `10`, supplied by the caller:

Example 12.1. Character Return Parameters

```
! Fortran function returns a character
CHARACTER*(*) FUNCTION CHF(C1,I)
   CHARACTER*(*) C1
   INTEGER I
END
```

```
/* C declaration of Fortran function */
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

If the Fortran function is declared to return a character value of constant length, for example `CHARACTER*4 FUNCTION CHF()`, the second extra parameter representing the length must still be supplied, but is not used.

Note

The value of the character function is not automatically NULL-terminated.

Complex Return Values

When a Fortran function returns a complex value, an argument needs to be added at the beginning of the C/C++ calling function's argument list; this argument is the address of the complex return value. Example 12.2, "COMPLEX Return Values" illustrates the extra parameter, `cplx`, supplied by the caller.

Example 12.2. COMPLEX Return Values

```
COMPLEX FUNCTION CF(C, I)
   INTEGER I
 . . .
END
```

```
extern void cf_();
typedef struct {float real, imag;} cplx;
cplx c1;
int i;
cf_(&c1, &i);
```

Array Indices

C/C++ arrays and Fortran arrays use different default initial array index values. By default, C/C++ arrays start at 0 and Fortran arrays start at 1. If you adjust your array comparisons so that a Fortran second element is compared to a C/C++ first element, and adjust similarly for other elements, you should not have problems working with this difference. If this is not satisfactory, you can declare your Fortran arrays to start at zero.

Another difference between Fortran and C/C++ arrays is the storage method used. Fortran uses column-major order and C/C++ use row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can

simply be reversed. For arrays other than single dimensional arrays, and square two-dimensional arrays, inter-language function mixing is not recommended.

Examples

This section contains examples that illustrate inter-language calling.

Example - Fortran Calling C

> **Note**
>
> There are other solutions to calling C from Fortran than the one presented in this section. For example, you can use the `iso_c_binding` intrinsic module which PGI does support. For more information on this module and for examples of how to use it, search the web using the keyword iso_c_binding.

Example 12.4, "C function f2c_func_" shows a C function that is called by the Fortran main program shown in Example 12.3, "Fortran Main Program f2c_main.f". Notice that each argument is defined as a pointer, since Fortran passes by reference. Also notice that the C function name uses all lower-case and a trailing "_".

Example 12.3. Fortran Main Program f2c_main.f

```
logical*1 bool1
 character letter1
 integer*4 numint1, numint2
 real numfloat1
 double precision numdoub1
 integer*2 numshor1
 external f2c_func

 call f2c_func_(bool1, letter1, numint1, numint2, numfloat1, numdoub1, numshor1)

 write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
 + bool1, letter1, numint1, numint2, numfloat1,numdoub1, numshor1

 end
```

Example 12.4. C function f2c_func_

```
#define TRUE 0xff
#define FALSE 0
void f2c_func_( bool1, letter1, numint1, numint2, numfloat1,\
 numdoub1, numshor1, len_letter1)
 char *bool1, *letter1;
 int *numint1, *numint2;
 float *numfloat1;
 double *numdoub1;
 short *numshor1;
 int len_letter1;
{
 *bool1 = TRUE;  *letter1 = 'v';
 *numint1 = 11;  *numint2 = -44;
 *numfloat1 = 39.6 ;
 *numdoub1 = 39.2;
 *numshor1 = 981;
}
```

Compile and execute the program `f2c_main.f` with the call to `f2c_func_` using the following command lines:

```
$ pgcc -c f2c_func.c
$ pgf95 f2c_func.o f2c_main.f
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

Example - C Calling Fortran

Example 12.5, "C Main Program c2f_main.c" shows a C main program that calls the Fortran subroutine shown in Example 12.6, "Fortran Subroutine c2f_sub.f". Notice that each call uses the & operator to pass by reference. Also notice that the call to the Fortran subroutine uses all lower-case and a trailing "_".

Example 12.5. C Main Program c2f_main.c

```
void main () {
 char bool1, letter1;
 int numint1, numint2;
 float numfloat1;
 double numdoub1;
 short numshor1;
 extern void c2f_func_();
 c2f_sub_(&bool1,&letter1,&numint1,&numint2,&numfloat1,&numdoub1,&numshor1, 1);
 printf(" %s %c %d %d %3.1f %.0f %d\n",
 bool1?"TRUE":"FALSE", letter1, numint1, numint2,
 numfloat1, numdoub1, numshor1);
}
```

Example 12.6. Fortran Subroutine c2f_sub.f

```
subroutine c2f_func ( bool1, letter1, numint1, numint2,
+ numfloat1, numdoub1, numshor1)
   logical*1 bool1
   character letter1
   integer numint1, numint2
   double precision numdoub1
   real numfloat1
   integer*2 numshor1

   bool1 = .true.
   letter1 = "v"
    numint1 = 11
    numint2 = -44
    numdoub1 = 902
    numfloat1 = 39.6
    numshor1 = 299
    return
end
```

To compile this Fortran subroutine and C program, use the following commands:

```
$ pgcc -c c2f_main.c
$ pgf95 -Mnomain c2f_main.o c2_sub.f
```

Executing the resulting `a.out` file should produce the following output:

```
TRUE v 11 -44 39.6 902 299
```

Example - C++ Calling C

Example 12.7, "C++ Main Program cp2c_main.C Calling a C Function" shows a C++ main program that calls the C function shown in Example 12.8, "Simple C Function c2cp_func.c".

Example 12.7. C++ Main Program cp2c_main.C Calling a C Function

```
extern "C" void cp2c_func(int n, int m, int *p);
#include <iostream>
main()
{
 int a,b,c;
 a=8;
 b=2;
 c=0;
 cout << "main: a = "<<a<<" b = "<<b<<"ptr c = "<<hex<<&c<< endl;
 cp2c_func(a,b,&c);
 cout << "main: res = "<<c<<endl;
}
```

Example 12.8. Simple C Function c2cp_func.c

```
void cp2c_func(num1, num2, res)
int num1, num2, *res;
{
 printf("func: a = %d b = %d ptr c = %x\n",num1,num2,res);
 *res=num1/num2;
 printf("func: res = %d\n",*res);
}
```

To compile this C function and C++ main program, use the following commands:

```
$ pgcc -c cp2c_func.c
$ pgcpp cp2c_main.C cp2c_func.o
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbffffb94
func: a = 8 b = 2 ptr c = bffffb94
func: res = 4
main: res = 4
```

Example - C Calling C++

Example 12.9, "C Main Program c2cp_main.c Calling a C++ Function" shows a C main program that calls the C++ function shown in Example 12.10, "Simple C++ Function c2cp_func.C with Extern C".

Example 12.9. C Main Program c2cp_main.c Calling a C++ Function

```
extern void c2cp_func(int a, int b, int *c);
#include <stdio.h>
main() {
 int a,b,c;
 a=8; b=2;
 printf("main: a = %d b = %d ptr c = %x\n",a,b,&c);
 c2cp_func(a,b,&c);
 printf("main: res = %d\n",c);
}
```

Example 12.10. Simple C++ Function c2cp_func.C with Extern C

```
#include <iostream>
extern "C" void c2cp_func(int num1,int num2,int *res)
{
 cout << "func: a = "<<num1<<" b = "<<num2<<"ptr c ="<<res<<endl;
 *res=num1/num2;
 cout << "func: res = "<<res<<endl;
}
```

To compile this C function and C++ main program, use the following commands:

```
$ pgcc -c c2cp_main.c
$ pgcpp c2cp_main.o c2cp_func.C
```

Executing the resulting a.out file should produce the following output:

```
main: a = 8 b = 2 ptr c = 0xbffffb94
func: a = 8 b = 2 ptr c = bffffb94
func: res = 4
main: res = 4
```

Note

You cannot use the extern "C" form of declaration for an object's member functions.

Example - Fortran Calling C++

The Fortran main program shown in Example 12.11, "Fortran Main Program f2cp_main.f calling a C++ function" calls the C++ function shown in Example 12.12, "C++ function f2cp_func.C". Notice that each argument is defined as a pointer in the C++ function, since Fortran passes by reference. Also notice that the C ++ function name uses all lower-case and a trailing "_":

Example 12.11. Fortran Main Program f2cp_main.f calling a C++ function

```
  logical*1 bool1
   character letter1
   integer*4 numint1, numint2
   real numfloat1
   double precision numdoub1
   integer*2 numshor1
 external f2cpfunc
 call f2cp_func (bool1, letter1, numint1,
 + numint2, numfloat1, numdoub1, numshor1)
   write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
   + bool1, letter1, numint1, numint2, numfloat1,
   + numdoub1, numshor1
 end
```

Example 12.12. C++ function f2cp_func.C

```
#define TRUE 0xff
#define FALSE 0
extern "C"
{
extern void f2cp_func_ (
 char *bool1, *letter1,
 int *numint1, *numint2,
```

```
 float *numfloat1,
 double *numdoub1,
 short *numshort1,
 int len_letter1)
{
 *bool1 = TRUE;     *letter1 = 'v';
 *numint1 = 11;     *numint2 = -44;
 *numfloat1 = 39.6; *numdoub1 = 39.2;   *numshort1 = 981;
}
}
```

Assuming the Fortran program is in a file `fmain.f`, and the C++ function is in a file `cpfunc.c`, create an executable, using the following command lines:

```
$ pgcpp -c f2cp_func.C
$ pgf95 f2cp_func.o f2cp_main.f -pgcpplibs
```

Executing the `a.out` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

Example - C++ Calling Fortran

Example 12.14, "Fortran Subroutine cp2f_func.f" shows a Fortran subroutine called by the C++ main program shown in Example 12.13, "C++ main program cp2f_main.C". Notice that each call uses the & operator to pass by reference. Also notice that the call to the Fortran subroutine uses all lower-case and a trailing "_":

Example 12.13. C++ main program cp2f_main.C

```
#include <iostream>
extern "C" { extern void cp2f_func_(char *,char *,int *,int *,
 float *,double *,short *); }
main ()
{
 char bool1, letter1;
 int numint1, numint2;
 float numfloat1;
 double numdoub1;
 short numshor1;

 cp2f_func(&bool1,&letter1,&numint1,&numint2,&numfloat1, &numdoub1,&numshor1);
 cout << " bool1 = ";
 bool1?cout << "TRUE ":cout << "FALSE "; cout <<endl;
 cout << " letter1 = " << letter1 <<endl;
 cout << " numint1 = " << numint1 <<endl;
 cout << " numint2 = " << numint2 <<endl;
 cout << " numfloat1 = " << numfloat1 <<endl;
 cout << " numdoub1 = " << numdoub1 <<endl;
 cout << " numshor1 = " << numshor1 <<endl;
}
```

Example 12.14. Fortran Subroutine cp2f_func.f

```
 subroutine cp2f_func ( bool1, letter1, numint1,
 + numint2, numfloat1, numdoub1, numshor1)
 logical*1 bool1
 character letter1
 integer numint1, numint2
```

```
double precision numdoub1
real numfloat1
integer*2 numshor1
bool1 = .true. ; letter1 = "v"
numint1 = 11 ; numint2 = -44
numdoub1 = 902 ; numfloat1 = 39.6 ; numshor1 = 299
return
end
```

To compile this Fortran subroutine and C++ program, use the following command lines:

```
$ pgf95 -c cp2f_func.f
$ pgcpp cp2f_func.o cp2f_main.C -pgf90libs
```

Executing this C++ main should produce the following output:

```
bool1 = TRUE
letter1 = v
numint1 = 11
numint2 = -44
numfloat1 = 39.6
numdoub1 = 902
numshor1 = 299
```

Note that you must explicitly link in the PGF95 runtime support libraries when linking pgf95-compiled program units into C or C++ main programs. When linking pgf77-compiled program units into C or C++ main programs, you need only link in `-lpgftnrtl`.

Win32 Calling Conventions

A calling convention is a set of conventions that describe the manner in which a particular routine is executed. A routine's calling conventions specify where parameters and function results are passed. For a stack-based routine, the calling conventions determine the structure of the routine's stack frame.

The calling convention for C/C++ is identical between most compilers on Win32 , Linux, Mac OS X, and Win64. However, Fortran calling conventions vary widely between legacy Win32 Fortran compilers and Linux or Win64 Fortran compilers.

Win32 Fortran Calling Conventions

Four styles of calling conventions are supported using the PGI Fortran compilers for Win32: Default, C, STDCALL, and UNIX.

- **Default** - Used in the absence of compilation flags or directives to alter the default.
- **C or STDCALL** - Used if an appropriate compiler directive is placed in a program unit containing the call. The C and STDCALL conventions are typically used to call routines coded in C or assembly language that depend on these conventions.
- **UNIX** - Used in any Fortran program unit compiled using the `-Miface=unix` (or `-Munix`) compilation flag.

The following table outlines each of these calling conventions.

Table 12.3. Calling Conventions Supported by the PGI Fortran Compilers

Convention	Default	STDCALL	C	UNIX
Case of symbol name	Upper	Lower	Lower	Lower
Leading underscore	Yes	Yes	Yes	Yes
Trailing underscore	No	No	No	Yes
Argument byte count added	Yes	Yes	No	No
Arguments passed by reference	Yes	No*	No*	Yes
Character argument length passed	After each char argument	No	No	End of argument list
First character of character string and passed by value	No	Yes	Yes	No
varargs support	No	No	Yes	Yes
Caller cleans stack	No	No	Yes	Yes

* Except arrays, which are always passed by reference even in the STDCALL and C conventions

Note

While it is compatible with the Fortran implementations of Microsoft and several other vendors, the C calling convention supported by the PGI Fortran compilers for Windows is not strictly compatible with the C calling convention used by most C/C++ compilers. In particular, symbol names produced by PGI Fortran compilers using the C convention are all lower case. The standard C convention is to preserve mixed-case symbol names. You can cause any of the PGI Fortran compilers to preserve mixed-case symbol names using the `-Mupcase` option, but be aware that this could have other ramifications on your program.

Symbol Name Construction and Calling Example

This section presents an example of the rules outlined in Table 12.3, "Calling Conventions Supported by the PGI Fortran Compilers," on page 159. In the pseudocode shown in the following examples, `%addr` refers to the address of a data item while `%val` refers to the value of that data item. Subroutine and function names are converted into symbol names according to the rules outlined in Table 12.3.

Consider the following subroutine call, where a is a double precision scalar, b is a real vector of size n, and n is an integer:

```
call work ( 'ERR', a, b, n)
```

- **Default** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all upper case, and appending an @ sign followed by an integer indicating the total number of bytes occupied by the argument list. Byte counts for character arguments appear immediately following the corresponding argument in the argument list.

  The following example is pseudocode for the preceding subroutine call using Default conventions:

```
call _WORK@20 (%addr('ERR'), 3, %addr(a), %addr(b), %addr(n))
```

- **STDCALL** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all lower case, and appending an @ sign followed by an integer indicating the total number of bytes occupied by the argument list. Character strings are truncated to the first character in the string, which is passed by value as the first byte in a 4-byte word. The following is an example of the pseudocode for the work subroutine call using STDCALL conventions:

  ```
  call _work@20 (%val('E'), %val(a), %addr(b), %val(n))
  ```

  Notice in this case that there are still 20 bytes in the argument list. However, rather than five 4-byte quantities as in the Default convention, there are three 4-byte quantities and one 8-byte quantity (the double precision value of a).

- **C** - The symbol name for the subroutine is constructed by pre-pending an underscore and converting to all lower case. Character strings are truncated to the first character in the string, which is passed by value as the first byte in a 4-byte word. The following is an example of the pseudocode for the work subroutine call using C conventions:

  ```
  call _work (%val('E'), %val(a), %addr(b), %val(n))
  ```

- **UNIX** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all lower case, and appending an underscore. Byte counts for character strings appear in sequence following the last argument in the argument list. The following is an example of the pseudocode for the work subroutine call using UNIX conventions:

  ```
  call _work_ (%addr('ERR'), %addr(a), %addr(b), %addr(n),3)
  ```

Using the Default Calling Convention

The Default calling convention is used if no directives are inserted to modify calling conventions and if neither the `-Miface=unix` (or `-Munix`) compilation flag is used. Refer to "Symbol Name Construction and Calling Example," on page 159 for a complete description of the Default calling convention.

Using the STDCALL Calling Convention

Using the STDCALL calling convention requires the insertion of a compiler directive into the declarations section of any Fortran program unit which calls the STDCALL program unit. This directive has no effect when either the `-Miface=unix` (or `-Munix`) compilation flag is used, meaning you cannot mix UNIX-style argument passing and STDCALL calling conventions within the same file. In the following example syntax for the directive, `work` is the name of the subroutine to be called using STDCALL conventions:

```
!DEC$ ATTRIBUTES STDCALL :: work
```

You can list more than one subroutine, separating them by commas. Refer to "Symbol Name Construction and Calling Example," on page 159 for a complete description of the implementation of STDCALL.

Note

- The directive prefix !DEC$ requires a space between the prefix and the directive keyword ATTRIBUTES.
- The ! must begin the prefix when compiling using Fortran 90 freeform format.

- The characters C or * can be used in place of ! in either form of the prefix when compiling used fixed-form format.
- The directives are completely case insensitive.

Using the C Calling Convention

Using the C calling convention requires the insertion of a compiler directive into the declarations section of any Fortran program unit which calls the C program unit. This directive has no effect when the `-Miface=unix` (or `-Munix`) compilation flag is used, meaning you cannot mix UNIX-style argument passing and C calling conventions within the same file.

Syntax for the directive is as follows:

```
!DEC$ ATTRIBUTES C :: work
```

Where `work` is the name of the subroutine to be called using C conventions. More than one subroutine may be listed, separated by commas. Refer to "Symbol Name Construction and Calling Example," on page 159 for a complete description of the implementation of the C calling convention.

Using the UNIX Calling Convention

Using the UNIX calling convention is straightforward. Any program unit compiled using `-Miface=unix` or the `-Munix` compilation flag uses the UNIX convention.

Using the CREF Calling Convention

Using the CREF calling convention is straightforward. Any program unit compiled using `-Miface=cref` compilation flag uses the CREF convention.

Chapter 13. Programming Considerations for 64-Bit Environments

PGI provides 64-bit compilers for the 64-bit Linux, Windows, SUA, and Mac OS X operating systems running on the x64 architecture. You can use these compilers to create programs that use 64-bit memory addresses. However, there are limitations to how this capability can be applied. With the exception of Linux86-64, the object file formats on all of the operating systems limit the total cumulative size of code plus static data to 2GB. This limit includes the code and statically declared data in the program and in system and user object libraries. Linux86-64 implements a mechanism that overcomes this limitations, as described in"Large Static Data in Linux," on page 164. This chapter describes the specifics of how to use the PGI compilers to make use of 64-bit memory addressing.

The 64-bit Windows, Linux, Mac OS X, and SUA environments maintain 32-bit compatibility, which means that 32-bit applications can be developed and executed on the corresponding 64-bit operating system.

> **Note**
>
> The 64-bit PGI compilers are 64-bit applications which cannot run on anything but 64-bit CPUs running 64-bit Operating Systems.

This chapter describes how to use the following options related to 64-bit programming.

`-fPIC`	`-mcmodel=medium`	`-Mlarge_arrays`
`-i8`	`-Mlargeaddressaware`	`-tp`

Data Types in the 64-Bit Environment

The size of some data types can be different in a 64-bit environment. This section describes the major differences. Refer to Chapter 15, "*Fortran, C, and C++ Data Types*" for detailed information.

C/C++ Data Types

On 32-bit Windows, int is 4 bytes, long is 4 bytes, and pointers are 4 bytes. On 64-bit windows, the size of an int is 4 bytes, a long is 4 bytes, and a pointer is 8 bytes.

On the 32-bit Linux, SUA, and Mac OS X operating systems, the size of an int is 4 bytes, a long is 4 bytes, and a pointer is 4 bytes. On the 64-bit Linux, SUA, and Mac OS X operating systems, the size of an int is 4 bytes, a long is 8 bytes, and a pointer is 8 bytes.

Fortran Data Types

In Fortran, the default size of the INTEGER type is 4 bytes. The `-i8` compiler option may be used to make the default size of all INTEGER data in the program 8 bytes.

When using the `-Mlarge_arrays` option, described in "64-Bit Array Indexing," on page 164, any 4-byte INTEGER variables that are used to index arrays are silently promoted by the compiler to 8 bytes. This promotion can lead to unexpected consequences, so 8-byte INTEGER variables are recommended for array indexing when using the option `-Mlarge_arrays`.

Large Static Data in Linux

Linux86-64 operating systems support two different memory models. The default model used by PGI compilers is the small memory model, which can be specified using -mcmodel=small. This is the 32-bit model, which limits the size of code plus statically allocated data, including system and user libraries, to 2GB. The medium memory model, specified by -mcmodel=medium, allows combined code and static data areas (.text and .bss sections) larger than 2GB. The `-mcmodel=medium` option must be used on both the compile command and the link command in order to take effect.

The Win64, SUA64, and 64-bit Mac OS X operating systems do not have any support for large static data declarations.

There are two drawbacks to using `-mcmodel=medium`. First, there is increased addressing overhead to support the large data range. This can affect performance, though the compilers seek to minimize the added overhead through careful instruction generation. Second, -mcmodel=medium cannot be used for objects in shared libraries, because there is no OS support for 64-bit dynamic linkage.

Large Dynamically Allocated Data

Dynamically allocated data objects in programs compiled by the 64-bit PGI compilers can be larger than 2GB. No special compiler options are required to enable this functionality. The size of the allocation is only limited by the system. However, to correctly access dynamically allocated arrays with more than 2G elements you should use the `-Mlarge_arrays` option, described in the following section.

64-Bit Array Indexing

The 64-bit PGI compilers provide an option, `-Mlarge_arrays`, that enables 64-bit indexing of arrays. This means that, as necessary, 64-bit INTEGER constants and variables are used to index arrays.

Note

In the presence of `-Mlarge_arrays`, the compiler may silently promote 32-bit integers to 64 bits, which can have unexpected side effects.

On Linux86-64, the `-Mlarge_arrays` option also enables single static data objects larger than 2 GB. This option is the default in the presence of `-mcmodel=medium`.

Note

On Win64, static data may not be larger than 2GB.

Compiler Options for 64-bit Programming

The usual switches that apply to 64-bit programmers seeking to increase the data range of their applications are in the following table.

Table 13.1. 64-bit Compiler Options

Option	Purpose	Considerations
–mcmodel=medium	Enlarge object size; Allow for declared data the size of larger than 2GB	Linux86-64 only. Slower execution. Cannot be used with –fPIC. Objects cannot be put into shared libraries.
–Mlargeaddressaware	[Win64 only] Generates code that allows for addresses greater than 2GB, using RIP-relative addressing.	Use –Mlargeaddressaware=no for a direct addressing mechanism that restricts the total addressable memory. This is not applicable if the object file is placed in a DLL. Further, if an object file is compiled with this option, it must also be used when linking.
–Mlarge_arrays	Perform all array-location-to-address calculations using 64-bit integer arithmetic.	Slightly slower execution. Is implicit with –mcmodel=medium. Can be used with option –mcmodel=small. Win64 does not support –Mlarge_arrays for static objects larger than 2GB.
–fPIC	Position independent code. Necessary for shared libraries.	Dynamic linking restricted to a 32-bit offset. External symbol references should refer to other shared lib routines, rather than the program calling them.
–i8	All INTEGER functions, data, and constants not explicitly declared INTEGER*4 are assumed to be INTEGER*8.	Users should take care to explicitly declare INTEGER functions as INTEGER*4.

The following table summarizes the limits of these programming models:

Table 13.2. Effects of Options on Memory and Array Sizes

Combined Compiler Options	Addr. Math		Max Size Gbytes			Comments
	A	I	AS	DS	TS	
`-tp k8-32` or `-tp p7`	32	32	2	2	2	32-bit linux86 programs
`-tp k8-64` or `-tp p7-64`	64	32	2	2	2	64-bit addr limited by option `-mcmodel=small`
`-tp k8-64 -fpic` or `-tp p7-64 -fpic`	64	32	2	2	2	`-fpic` *incompatible with* –mcmodel=medium
`-tp k8-64` or `-tp p7-64 -mcmodel=medium`	64	64	>2	>2	>2	Enable full support for 64-bit data addressing

Column Legend	
A	Address Type - size in bits of data used for address calculations, 32-bit or 64-bit.
I	Index Arithmetic - bit-size of data used to index into arrays and other aggregate data structures. If 32-bit, total range of any single data object is limited to 2GB.
AS	Maximum Array Size - the maximum size in gigabytes of any single data object.
DS	*Maximum Data Size* - max size in gigabytes combined of all data objects in .bss
TS	Maximum Total Size - max size in gigabytes, in aggregate, of all executable code and data objects in a running program.

Practical Limitations of Large Array Programming

The 64-bit addressing capability of the Linux86-64 and Win64 environments can cause unexpected issues when data sizes are enlarged significantly. The following table describes the most common occurrences of practical limitations of large array programming.

Table 13.3. 64-Bit Limitations

array initialization	Initializing a large array with a data statement may result in very large assembly and object files, where a line of assembler source is required for each element in the initialized array. Compilation and linking can be very time consuming as well. To avoid this issue, consider initializing large arrays in a loop at runtime rather than in a data statement.

stack space	Stack space can be a problem for data that is stack-based. In Win64, stack space can be increased by using this link-time switch, where N is the desired stack size:`–Wl,-stack:N`
	Note
	In linux86-64, stack size is increased in the environment. Setting stacksize to unlimited often is not large enough.
	`limit stacksize new_size ! in csh`
	`ulimit –s new_size ! in bash`

page swapping	If your executable is much larger than the physical size of memory, page swapping can cause it to run dramatically slower; it may even fail. This is not a compiler problem. Try smaller data sets to determine whether or not a problem is due to page thrashing.
configured space	Be sure your linux86-64 system is configured with swap space sufficiently large to support the data sets used in your application(s). If your memory +swap space is not sufficiently large, your application will likely encounter a segmentation fault at runtime.
support for large address offsets in object file format	Arrays that are not dynamically allocated are limited by how the compiler can express the 'distance' between them when generating code. A field in the object file stores this 'distance' value, which is limited to 32-bits on Win32, Win64, linux86, and linux86-64 with –mcmodel=small. It is 64-bits on linux86-64 with –mcmodel=medium. **Note.** Without the 64-bit offset support in the object file format, large arrays cannot be declared statically or locally stack-based.

Medium Memory Model and Large Array in C

Example 13.1. Medium Memory Model and Large Array in C

Consider the following example, where the aggregate size of the arrays exceeds 2GB.

```
% cat bigadd.
#include <stdio.h>
#define SIZE 600000000 /* > 2GB/4 */
static float a[SIZE], b[SIZE];
int
main()
{
 long long i, n, m;
 float c[SIZE]; /* goes on stack */
 n = SIZE;
 m = 0;
 for (i = 0; i < n; i += 10000) {
 a[i] = i + 1;
 b[i] = 2.0 * (i + 1);
 c[i] = a[i] + b[i];
 m = i;
 }
 printf("a[0]=%g b[0]=%g c[0]=%g\n", a[0], b[0], c[0]);
```

```
 printf("m=%lld a[%lld]=%g b[%lld]=%gc[%lld]=%g\n",m,m,a[m],m,b[m],m,c[m]);
 return 0;
}
```

```
% pgcc –mcmodel=medium –o bigadd bigadd.c
```

When SIZE is greater than 2G/4, and the arrays are of type float with 4 bytes per element, the size of each array is greater than 2GB. With pgcc, using the –mcmodel=medium switch, a static data object can now be > 2GB in size. If you execute with these settings in your environment, you may see the following:

```
% bigadd
 Segmentation fault
```

Execution fails because the stack size is not large enough. You can most likely correct this error by using the **limit stacksize** command to reset the stack size in your environment:

```
% limit stacksize 3000M
```

> **Note**
>
> The command **limit stacksize unlimited** probably does not provide as large a stack as we are using in the Example 13.1.

```
% bigadd
a[0]=1 b[0]=2 c[0]=3
n=599990000 a[599990000]=5.9999e+08 b[599990000]=1.19998e+09
c[599990000]=1.79997e+09
```

The size of the bss section of the bigadd executable is now larger than 2GB:

```
% size –-format=sysv bigadd | grep bss
.bss 4800000008 5245696
% size --format=sysv bigadd | grep Total
Total 4800005080
```

Medium Memory Model and Large Array in Fortran

The following example works with both the PGF95 and PGF77 compilers included in Release 7.0. Both compilers use 64-bit addresses and index arithmetic when the –mcmodel=medium option is used.

Example 13.2. Medium Memory Model and Large Array in Fortran

Consider the following example:

```
% cat mat.f
 program mat
 integer i, j, k, size, l, m, n parameter (size=16000) ! >2GB
 parameter (m=size,n=size)
 real*8 a(m,n),b(m,n),c(m,n),d
 do i = 1, m
 do j = 1, n
 a(i,j)=10000.0D0*dble(i)+dble(j)
 b(i,j)=20000.0D0*dble(i)+dble(j)
 enddo
 enddo
!$omp parallel
!$omp do
```

```
 do i = 1, m
 do j = 1, n
 c(i,j) = a(i,j) + b(i,j)
 enddo
 enddo
!$omp do
 do i=1,m
 do j = 1, n
 d = 30000.0D0*dble(i)+dble(j)+dble(j)
 if(d .ne. c(i,j)) then
 print *,"err i=",i,"j=",j
 print *,"c(i,j)=",c(i,j)
 print *,"d=",d
 stop
 endif
 enddo
 enddo
!$omp end parallel
 print *, "M =",M,", N =",N
 print *, "c(M,N) = ", c(m,n)
 end
```

When compiled with the PGF95 compiler using –mcmodel=medium:

```
% pgf95 –mp –o mat mat.f –i8 –mcmodel=medium
% setenv OMP_NUM_THREADS 2
% mat
M = 16000 , N = 16000
c(M,N) = 480032000.0000000
```

Large Array and Small Memory Model in Fortran

The following example uses large, dynamically-allocated arrays. The code is divided into a main and subroutine so you could put the subroutine into a shared library. Dynamic allocation of large arrays saves space in the size of executable and saves time initializing data. Further, the routines can be compiled with 32-bit compilers, by just decreasing the parameter size.

Example 13.3. Large Array and Small Memory Model in Fortran

```
% cat mat_allo.f90
program mat_allo
 integer i, j
 integer size, m, n
 parameter (size=16000)
 parameter (m=size,n=size)
 double precision, allocatable::a(:,:),b(:,:),c(:,:)
 allocate(a(m,n), b(m,n), c(m,n))
 do i = 100, m, 1
 do j = 100, n, 1
 a(i,j) = 10000.0D0 * dble(i) + dble(j)
 b(i,j) = 20000.0D0 * dble(i) + dble(j)
 enddo
 enddo
 call mat_add(a,b,c,m,n)
 print *, "M =",m,",N =",n
 print *, "c(M,N) = ", c(m,n)
end
```

```
subroutine mat_add(a,b,c,m,n)
 integer m, n, i, j
 double precision a(m,n),b(m,n),c(m,n)
!$omp do
 do i = 1, m
 do j = 1, n
 c(i,j) = a(i,j) + b(i,j)
 enddo
 enddo
 return
end
% pgf95 –o mat_allo mat_allo.f90 –i8 –Mlarge_arrays -mp -fast
```

Chapter 14. C/C++ Inline Assembly and Intrinsics

Inline Assembly

Inline Assembly lets you specify machine instructions inside a "C" function. The format for an inline assembly instruction is this:

```
{ asm | __asm__ } ("string");
```

The asm statement begins with the *asm* or *__asm__* keyword. The __asm__ keyword is typically used in header files that may be included in ISO "C" programs.

"*string*" is one or more machine specific instructions separated with a semi-colon (*;*) or newline (*\n*) character. These instructions are inserted directly into the compiler's assembly-language output for the enclosing function.

Some simple asm statements are:

```
asm ("cli");
asm ("sti");
```

These asm statements disable and enable system interrupts respectively.

In the following example, the eax register is set to zero.

```
asm( "pushl %eax\n\t" "movl $0, %eax\n\t" "popl %eax");
```

Notice that eax is pushed on the stack so that it is it not clobbered. When the statement is done with eax, it is restored with the popl instruction.

Typically a program uses macros that enclose asm statements. The following two examples use the interrupt constructs created previously in this section:

```
#define disableInt __asm__ ("cli");
#define enableInt __asm__ ("sti");
```

Extended Inline Assembly

"Inline Assembly," on page 171 explains how to use inline assembly to specify machine specific instructions inside a "C" function. This approach works well for simple machine operations such as disabling and enabling system interrupts. However, inline assembly has three distinct limitations:

1. The programmer must choose the registers required by the inline assembly.
2. To prevent register clobbering, the inline assembly must include push and pop code for registers that get modified by the inline assembly.
3. There is no easy way to access stack variables in an inline assembly statement.

Extended Inline Assembly was created to address these limitations. The format for extended inline assembly, also known as *extended asm*, is as follows:

```
{ asm | __asm__ } [ volatile | __volatile__ ]
("string" [: [output operands]] [: [input operands]] [: [clobberlist]]);
```

- Extended asm statements begin with the *asm* or *__asm__* keyword. Typically the *__asm__* keyword is used in header files that may be included by ISO "C" programs.
- An optional *volatile* or *__volatile__* keyword may appear after the *asm* keyword. This keyword instructs the compiler not to delete, move significantly, or combine with any other asm statement. Like __asm__, the __volatile__ keyword is typically used with header files that may be included by ISO "C" programs.
- "*string*" is one or more machine specific instructions separated with a semi-colon (*;*) or newline (*\n*) character. The string can also contain operands specified in the *[output operands]*, *[input operands]*, and *[clobber list]*. The instructions are inserted directly into the compiler's assembly-language output for the enclosing function.
- The *[output operands]*, *[input operands]*, and *[clobber list]* items each describe the effect of the instruction for the compiler. For example:

  ```
  asm( "movl %1, %%eax\n" "movl %%eax, %0":"=r" (x) : "r" (y) : "%eax" );
  ```

  where "=r" (x) is an output operand
  "r" (y) is an input operand.
  "%eax" is the clobber list consisting of one register, "%eax".

  The notation for the output and input operands is a constraint string surrounded by quotes, followed by an expression, and surrounded by parentheses. The constraint string describes how the input and output operands are used in the asm "string". For example, "r" tells the compiler that the operand is a register. The "=" tells the compiler that the operand is write only, which means that a value is stored in an output operand's expression at the end of the asm statement.

  Each operand is referenced in the asm "string" by a percent "%" and its number. The first operand is number 0, the second is number 1, the third is number 2, and so on. In the preceding example, "%0" references the output operand, and "%1" references the input operand. The asm "string" also contains "% %eax", which references machine register "%eax". Hard coded registers like "%eax" should be specified in the clobber list to prevent conflicts with other instructions in the compiler's assembly-language output.

[output operands], *[input operands]*, and *[clobber list]* items are described in more detail in the following sections.

Output Operands

The *[output operands]* are an optional list of output constraint and expression pairs that specify the result(s) of the asm statement. An output constraint is a string that specifies how a result is delivered to the expression. For example, "=r" (x) says the output operand is a write-only register that stores its value in the "C" variable x at the end of the asm statement. An example follows:

```
int x;
void example()
{
 asm( "movl $0, %0" : "=r" (x) );
}
```

The previous example assigns 0 to the "C" variable x. For the function in this example, the compiler produces the following assembly. If you want to produce an assembly listing, compile the example with the pgcc -S compiler option:

```
example:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 8
 movl $0, %eax
 movl %eax, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

In the generated assembly shown, notice that the compiler generated two statements for the asm statement at line number 5. The compiler generated "*movl $0, %eax*" from the asm "*string*". Also notice that *%eax* appears in place of "*%0*" because the compiler assigned the *%eax* register to variable *x*. Since item 0 is an output operand, the result must be stored in its expression (*x*). The instruction *movl %eax, x(%rip)* assigns the output operand to variable *x*.

In addition to write-only output operands, there are *read/write* output operands designated with a "+" instead of a "=". For example, "+*r*" (*x*) tells the compiler to initialize the output operand with variable x at the beginning of the asm statement.

To illustrate this point, the following example increments variable x by 1:

```
int x=1;
void example2()
{
 asm( "addl $1, %0" : "+r" (x) );
}
```

To perform the increment, the output operand must be initialized with variable x. The *read/write* constraint modifier ("+") instructs the compiler to initialize the output operand with its expression. The compiler generates the following assembly code for the example2() function:

```
example2:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 5
 movl x(%rip), %eax
 addl $1, %eax
 movl %eax, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

From the example2() code, two extraneous moves are generated in the assembly: one movl for initializing the output register and a second movl to write it to variable x. To eliminate these moves, use a memory constraint type instead of a register constraint type, as shown in the following example:

```
int x=1;
void example2()
{
 asm( "addl $1, %0" : "+m" (x) );
}
```

The compiler generates a memory reference in place of a memory constraint. This eliminates the two extraneous moves. Because the assembly uses a memory reference to variable x, it does not have to move x into a register prior to the asm statement; nor does it need to store the result after the asm statement. Additional constraint types are found in "Additional Constraints," on page 177.

```
example2:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 5
 addl $1, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

The examples thus far have used only one output operand. Because extended asm accepts a list of output operands, asm statements can have more than one result, as shown in the following example:

```
void example4()
{
int x=1; int y=2;
asm( "addl $1, %1\n" "addl %1, %0": "+r" (x), "+m" (y) );
}
```

This example increments variable *y* by *1* then adds it to variable *x*. Multiple output operands are separated with a comma. The first output operand is item 0 ("%0") and the second is item 1 ("%1") in the asm "*string*". The resulting values for *x* and *y* are *4* and *3* respectively.

Input Operands

The *[input operands]* are an optional list of input constraint and expression pairs that specify what "C" values are needed by the asm statement. The input constraints specify how the data is delivered to the asm statement. For example, "*r*" *(x)* says that the input operand is a register that has a copy of the value stored in "C" variable *x*. Another example is "*m*" *(x)* which says that the input item is the *memory* location associated with variable *x*. Other constraint types are discussed in "Additional Constraints," on page 177. An example follows:

```
void example5()
{
int x=1;
int y=2;
int z=3;
asm( "addl %2, %1\n" "addl %2, %0" : "+r" (x), "+m" (y) : "r" (z) );
}
```

The previous example adds variable z, item 2, to variable x and variable y. The resulting values for x and y are 4 and 5 respectively.

Another type of input constraint worth mentioning here is the *matching constraint*. A matching constraint is used to specify an operand that fills both an input as well as an output role. An example follows:

```
int x=1;
void example6()
{
asm( "addl $1, %1"
 : "=r" (x)
 : "0" (x) );
}
```

The previous example is equivalent to the *example2()* function shown in "Output Operands," on page 173. The constraint/expression pair, "*0*" *(x)*, tells the compiler to initialize output item *0* with variable *x* at the beginning of the *asm* statement. The resulting value for *x* is 2. Also note that "*%1*" in the asm "*string*" means the same thing as "*%0*" in this case. That is because there is only one operand with both an input and an output role.

Matching constraints are very similar to the *read/write* output operands mentioned in "Output Operands," on page 173. However, there is one key difference between *read/write* output operands and *matching constraints*. The *matching constraint* can have an *input expression* that differs from its *output expression*.

The following example uses different values for the input and output roles:

```
int x;
int y=2;
void example7()
{
asm( "addl $1, %1"
 : "=r" (x)
 : "0" (y) );
}
```

The compiler generates the following assembly for example7():

```
example7:
..Dcfb0:
 pushq %rbp
..Dcfi0:
```

```
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 8
 movl y(%rip), %eax
 addl $1, %eax
 movl %eax, x(%rip)
## lineno: 0
 popq %rbp
 ret
```

Variable *x* gets initialized with the value stored in *y*, which is *2*. After adding *1*, the resulting value for variable *x* is *3*.

Because *matching constraints* perform an input role for an output operand, it does not make sense for the output operand to have the read/write ("+") modifier. In fact, the compiler disallows *matching constraints* with read/write output operands. The output operand must have a write only ("=") modifier.

Clobber List

The *[clobber list]* is an optional list of strings that hold machine registers used in the asm "*string*". Essentially, these strings tell the compiler which registers may be clobbered by the asm statement. By placing registers in this list, the programmer does not have to explicitly save and restore them as required in traditional inline assembly (described in "Inline Assembly," on page 171). The compiler takes care of any required saving and restoring of the registers in this list.

Each machine register in the [clobber list] is a string separated by a comma. The leading '%' is optional in the register name. For example, "%eax" is equivalent to "eax". When specifying the register inside the asm "string", you must include two leading '%' characters in front of the name (for example., "%%eax"). Otherwise, the compiler will behave as if a bad input/output operand was specified and generate an error message. An example follows:

```
void example8()
{
int x;
int y=2;
asm( "movl %1, %%eax\n"
 "movl %1, %%edx\n"
 "addl %%edx, %%eax\n"
 "addl %%eax, %0"
 : "=r" (x)
 : "0" (y)
 : "eax", "edx" );
}
```

This code uses two hard-coded registers, eax and edx. It performs the equivalent of 3*y and assigns it to x, producing a result of 6.

In addition to machine registers, the clobber list may contain the following special flags:

"cc"
: The asm statement may alter the condition code register.

"memory"
: The asm statement may modify memory in an unpredictable fashion.

When the "memory" flag is present, the compiler does not keep memory values cached in registers across the asm statement and does not optimize stores or loads to that memory. For example:

```
asm("call MyFunc":::"memory");
```

This asm statement contains a "memory" flag because it contains a call. The callee may otherwise clobber registers in use by the caller without the "memory" flag.

The following function uses extended asm and the "cc" flag to compute a power of 2 that is less than or equal to the input parameter n.

```
#pragma noinline
int asmDivideConquer(int n)
{
 int ax = 0;
 int bx = 1;
 asm (
 "LogLoop:\n"
 "cmp %2, %1\n"
 "jnle Done\n"
 "inc %0\n"
 "add %1,%1\n"
 "jmp LogLoop\n"
 "Done:\n"
 "dec %0\n"
 :"+r" (ax), "+r" (bx) : "r" (n) : "cc");
 return ax;
}
```

The "cc" flag is used because the asm statement contains some control flow that may alter the condition code register. The #pragma noinline statement prevents the compiler from inlining the asmDivideConquer() function. If the compiler inlines asmDivideConquer(), then it may illegally duplicate the labels LogLoop and Done in the generated assembly.

Additional Constraints

Operand constraints can be divided into four main categories:

- Simple Constraints
- Machine Constraints
- Multiple Alternative Constraints
- Constraint Modifiers

Simple Constraints

The simplest kind of constraint is a string of letters or characters, known as *Simple Constraints*, such as the "r" and "m" constraints introduced in “Output Operands,” on page 173. Table 14.1, “Simple Constraints” describes these constraints.

Table 14.1. Simple Constraints

Constraint	Description
whitespace	Whitespace characters are ignored.
E	An immediate floating point operand.
F	Same as "E".
g	Any general purpose register, memory, or immediate integer operand is allowed.
i	An immediate integer operand.
m	A memory operand. Any address supported by the machine is allowed.
n	Same as "i".
o	Same as "m".
p	An operand that is a valid memory address. The expression associated with the constraint is expected to evaluate to an address (for example, "p" (&x)).
r	A general purpose register operand.
X	Same as "g".
0,1,2,..9	Matching Constraint. See “Input Operands,” on page 175 for a description.

The following example uses the general or "g" constraint, which allows the compiler to pick an appropriate constraint type for the operand; the compiler chooses from a general purpose register, memory, or immediate operand. This code lets the compiler choose the constraint type for "y".

```
void example9()
{
 int x, y=2;
 asm( "movl %1, %0\n" : "=r"
(x) : "g" (y) );
}
```

This technique can result in more efficient code. For example, when compiling example9() the compiler replaces the load and store of y with a constant 2. The compiler can then generate an immediate 2 for the y operand in the example. The assembly generated by pgcc for our example is as follows:

```
example9:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 3
 movl $2, %eax
## lineno: 6
 popq %rbp
 ret
```

In this example, notice the use of $2 for the "y" operand.

Of course, if y is always 2, then the immediate value may be used instead of the variable with the "i" constraint, as shown here:

```
void example10()
{
int x;
asm( "movl %1, %0\n"
 : "=r" (x)
 : "i" (2) );
}
```

Compiling example10() with pgcc produces assembly similar to that produced for example9().

Machine Constraints

Another category of constraints is *Machine Constraints*. The x86 and x86_64 architectures have several classes of registers. To choose a particular class of register, you can use the x86/x86_64 machine constraints described in Table 14.2, "x86/x86_64 Machine Constraints".

Table 14.2. x86/x86_64 Machine Constraints

Constraint	Description
a	a register (e.g., %al, %ax, %eax, %rax)
A	Specifies a or d registers. This is used primarily for holding 64-bit integer values on 32 bit targets. The d register holds the most significant bits and the a register holds the least significant bits.
b	b register (e.g, %bl, %bx, %ebx, %rbx)
c	c register (e.g., %cl, %cx, %ecx, %rcx)
C	Not supported.
d	d register (e.g., %dl, %dx, %edx, %rdx)
D	di register (e.g., %dil, %di, %edi, %rdi)
e	Constant in range of 0xffffffff to 0x7fffffff
f	Not supported.
G	Floating point constant in range of 0.0 to 1.0.
I	Constant in range of 0 to 31 (e.g., for 32-bit shifts).
J	Constant in range of 0 to 63 (e.g., for 64-bit shifts)
K	Constant in range of 0 to 127.
L	Constant in range of 0 to 65535.
M	Constant in range of 0 to 3 constant (e.g., shifts for lea instruction).
N	Constant in range of 0 to 255 (e.g., for out instruction).
q	Same as "r" simple constraint.
Q	Same as "r" simple constraint.
R	Same as "r" simple constraint.
S	si register (e.g., %sil, %si, %edi, %rsi)
t	Not supported.

Constraint	Description
u	Not supported.
x	XMM SSE register
y	Not supported.
Z	Constant in range of 0 to 0x7fffffff.

The following example uses the "x" or XMM register constraint to subtract c from b and store the result in a.

```
double example11()
{
 double a;
 double b = 400.99;
 double c = 300.98;
 asm ( "subpd %2, %0;"
 :"=x" (a)
 : "0" (b), "x" (c)
 );
 return a;
}
```

The generated assembly for this example is this:

```
example11:
..Dcfb0:
 pushq %rbp
..Dcfi0:
 movq %rsp, %rbp
..Dcfi1:
..EN1:
## lineno: 4
 movsd .C00128(%rip), %xmm1
 movsd .C00130(%rip), %xmm2
 movapd %xmm1, %xmm0
 subpd %xmm2, %xmm0;
## lineno: 10
## lineno: 11
 popq %rbp
 ret
```

If a specified register is not available, the pgcc and pgcpp compilers issue an error message. For example, pgcc and pgcpp reserves the "%ebx" register for Position Independent Code (PIC) on 32-bit system targets. If a program has an asm statement with a "b" register for one of the operands, the compiler will not be able to obtain that register when compiling for 32-bit with the -fPIC switch (which generates PIC).

To illustrate this point, the following example is compiled for a 32-bit target using PIC:

```
void example12()
{
int x=1;
int y=1;
asm( "addl %1, %0\n"
 : "+a" (x)
 : "b" (y) );
}
```

Compiling with the "-tp p7" switch chooses a 32-bit target.

```
% pgcc example12.c -fPIC -c -tp p7
PGC-S-0354-Can't find a register in class 'BREG' for extended ASM
operand 1 (example12.c: 3)
PGC/x86 Linux/x86 Rel Dev: compilation completed with severe errors
```

Multiple Alternative Constraints

Sometimes a single instruction can take a variety of operand types. For example, the x86 permits register-to-memory and memory-to-register operations. To allow this flexibility in inline assembly, use *multiple alternative constraints*. An *alternative* is a series of constraints for each operand.

To specify multiple alternatives, separate each alternative with a comma.

Table 14.3. Multiple Alternative Constraints

Constraint	Description
,	Separates each alternative for a particular operand.
?	Ignored
!	Ignored

The following example uses multiple alternatives for an add operation.

```
void example13()
{
int x=1;
int y=1;
asm( "addl %1, %0\n"
 : "+ab,cd" (x)
 : "db,cam" (y) );
}
```

example13() has two alternatives for each operand: "ab,cd" for the output operand and "db,cam" for the input operand. Each operand must have the same number of alternatives; however, each alternative can have any number of constraints (for example, the output operand in *example13()* has two constraints for its second alternative and the input operand has three for its second alternative).

The compiler first tries to satisfy the left-most alternative of the first operand (for example, the output operand in *example13()*). When satisfying the operand, the compiler starts with the left-most constraint. If the compiler cannot satisfy an alternative with this constraint (for example, if the desired register is not available), it tries to use any subsequent constraints. If the compiler runs out of constraints, it moves on to the next alternative. If the compiler runs out of alternatives, it issues an error similar to the one mentioned in *example12()*. If an alternative is found, the compiler uses the same alternative for subsequent operands. For example, if the compiler chooses the "c" register for the output operand in example13(), then it will use either the "a" or "m" constraint for the input operand.

Constraint Modifiers

Characters that affect the compiler's interpretation of a constraint are known as *Constraint Modifiers*. Two constraint modifiers, the "=" and the "+", were introduced in “Output Operands,” on page 173. Table 14.4 summarizes each constraint modifier.

Table 14.4. Constraint Modifier Characters

Constraint Modifier	Description
=	This operand is write-only. It is valid for output operands only. If specified, the "=" must appear as the first character of the constraint string.
+	This operand is both read and written by the instruction. It is valid for output operands only. The output operand is initialized with its expression before the first instruction in the asm statement. If specified, the "+" must appear as the first character of the constraint string.
&	A constraint or an alternative constraint, as defined in "Multiple Alternative Constraints," on page 181, containing an "&" indicates that the output operand is an *early clobber operand*. This type operand is an output operand that may be modified before the asm statement finishes using all of the input operands. The compiler will not place this operand in a register that may be used as an input operand or part of any memory address.
%	Ignored.
#	Characters following a "#" up to the first comma (if present) are to be ignored in the constraint.
*	The character that follows the "*" is to be ignored in the constraint.

The "=" and "+" modifiers apply to the operand, regardless of the number of alternatives in the constraint string. For example, the "+" in the output operand of example13() appears once and applies to both alternatives in the constraint string. The "&", "#", and "*" modifiers apply only to the alternative in which they appear.

Normally, the compiler assumes that input operands are used before assigning results to the output operands. This assumption lets the compiler reuse registers as needed inside the asm statement. However, if the asm statement does not follow this convention, the compiler may indiscriminately clobber a result register with an input operand. To prevent this behavior, apply the early clobber "&" modifier. An example follows:

```
void example15()
{
int w=1;
int z;
asm( "movl $1, %0\n"
 "addl %2, %0\n"
 "movl %2, %1"
 : "=a" (w), "=r" (z) : "r" (w) );
}
```

The previous code example presents an interesting ambiguity because "w" appears both as an output and as an input operand. So, the value of "z" can be either 1 or 2, depending on whether the compiler uses the same register for operand 0 and operand 2. The use of constraint "r" for operand 2 allows the compiler to pick any general purpose register, so it may (or may not) pick register "a" for operand 2. This ambiguity can be eliminated by changing the constraint for operand 2 from "r" to "a" so the value of "z" will be 2, or by adding an early clobber "&" modifier so that "z" will be 1. The following example shows the same function with an early clobber "&" modifier:

```
void example16()
{
int w=1;
int z;
asm( "movl $1, %0\n"
 "addl %2, %0\n"
 "movl %2, %1"
 : "=&a" (w), "=r" (z) : "r" (w) );
}
```

Adding the early clobber "&" forces the compiler not to use the "a" register for anything other than operand 0. Operand 2 will therefore get its own register with its own copy of "w". The result for "z" in *example16()* is 1.

Operand Aliases

Extended asm specifies operands in assembly strings with a percent '%' followed by the operand number. For example, "%0" references operand 0 or the output item "=&a" (w) in function *example16()* in the previous example. Extended asm also supports operand aliasing, which allows use of a symbolic name instead of a number for specifying operands, as illustrated in this example:

```
void example17()
{
int w=1, z=0;
asm( "movl $1, %[output1]\n"
 "addl %[input], %[output1]\n"
 "movl %[input], %[output2]"
 : [output1] "=&a" (w), [output2] "=r"
(z)
 : [input] "r" (w));
}
```

In *example17()*, "%[output1]" is an alias for "%0", "%[output2]" is an alias for "%1", and "%[input]" is an alias for "%2". Aliases and numeric references can be mixed, as shown in the following example:

```
void example18()
{
int w=1, z=0;
asm( "movl $1, %[output1]\n"
 "addl %[input], %0\n"
 "movl %[input], %[output2]"
 : [output1] "=&a" (w), [output2] "=r" (z)
 : [input] "r" (w));
}
```

In *example18()*, "%0" and "%[output1]" both represent the output operand.

Assembly String Modifiers

Special character sequences in the assembly string affect the way the assembly is generated by the compiler. For example, the "%" is an escape sequence for specifying an operand, "%%" produces a percent for hard coded registers, and "\n" specifies a new line. Table 14.5, "Assembly String Modifier Characters"summarizes these modifiers, known as *Assembly String Modifiers*.

Table 14.5. Assembly String Modifier Characters

Modifier	Description
\	Same as \ in printf format strings.
%*	Adds a '*' in the assembly string.
%%	Adds a '%' in the assembly string.
%A	Adds a '*' in front of an operand in the assembly string. (For example, %A0 adds a '*' in front of operand 0 in the assembly output.)
%B	Produces the byte op code suffix for this operand. (For example, %b0 produces 'b' on x86 and x86_64.)
%L	Produces the word op code suffix for this operand. (For example, %L0 produces 'l' on x86 and x86_64.)
%P	If producing Position Independent Code (PIC), the compiler adds the PIC suffix for this operand. (For example, %P0 produces @PLT on x86 and x86_64.)
%Q	Produces a quad word op code suffix for this operand if is supported by the target. Otherwise, it produces a word op code suffix. (For example, %Q0 produces 'q' on x86_64 and 'l' on x86.)
%S	Produces 's' suffix for this operand. (For example, %S0 produces 's' on x86 and x86_64.)
%T	Produces 't' suffix for this operand. (For example, %S0 produces 't' on x86 and x86_64.)
%W	Produces the half word op code suffix for this operand. (For example, %W0 produces 'w' on x86 and x86_64.)
%a	Adds open and close parentheses ( ) around the operand.
%b	Produces the byte register name for an operand. (For example, if operand 0 is in register 'a', then %b0 will produce '%al'.)
%c	Cuts the '$' character from an immediate operand.
%k	Produces the word register name for an operand. (For example, if operand 0 is in register 'a', then %k0 will produce '%eax'.)
%q	Produces the quad word register name for an operand if the target supports quad word. Otherwise, it produces a word register name. (For example, if operand 0 is in register 'a', then %q0 produces %rax on x86_64 or %eax on x86.)
%w	Produces the half word register name for an operand. (For example, if operand 0 is in register 'a', then %w0 will produce '%ax'.)
%z	Produces an op code suffix based on the size of an operand. (For example, 'b' for byte, 'w' for half word, 'l' for word, and 'q' for quad word.)
%+ %C %D %F %O %X %f %h %l %n %s %y are not supported.	

These modifiers begin with either a backslash "\" or a percent "%".

The modifiers that begin with a backslash "\" (e.g., "\n") have the same effect as they do in a printf format string. The modifiers that are preceded with a "%" are used to modify a particular operand.

These modifiers begin with either a backslash "\" or a percent "%" For example, "%b0" means, "produce the byte or 8 bit version of operand 0". If operand 0 is a register, it will produce a byte register such as %al, %bl, %cl, and so on.

Consider this example:

```
void example19()
{
int a = 1;
int *p = &a;
asm ("add%z0 %q1, %a0"
 : "=&p" (p) : "r" (a), "0" (p) );
}
```

On an x86 target, the compiler produces the following instruction for the asm string shown in the preceding example:

```
addl %ecx, (%eax)
```

The "%z0" modifier produced an 'l' (lower-case 'L') suffix because the size of pointer p is 32 bits on x86. The "%q1" modifier produced the word register name for variable a. The "%a0" instructs the compiler to add parentheses around operand 0, hence "(%eax)".

On an x86_64 target, the compiler produces the following instruction for the asm string shown in the preceding example:

```
addq %rcx, (%rax)
```

The "%z0" modifier produced a 'q' suffix because the size of pointer p is 64-bit on x86_64. Because x86_64 supports quad word registers, the "%q1" modifier produced the quad word register name (%rax) for variable a.

Extended Asm Macros

As with traditional inline assembly, described in"Inline Assembly," on page 171, extended asm can be used in a macro. For example, you can use the following macro to access the runtime stack pointer.

```
#define GET_SP(x) \
asm("mov %%sp, %0": "=m" (##x):: "%sp" );
void example20()
{
 void * stack_pointer;
 GET_SP(stack_pointer);
}
```

The GET_SP macro assigns the value of the stack pointer to whatever is inserted in its argument (for example, stack_pointer). Another "C" extension known as *statement expressions* is used to write the GET_SP macro another way:

```
#define GET_SP2 ({ \
void *my_stack_ptr; \
asm("mov %%sp, %0": "=m" (my_stack_ptr) :: "%sp" ); \
my_stack_ptr; \
})
```

```
void example21()
{
 void * stack_pointer = GET_SP2;
}
```

The statement expression allows a body of code to evaluate to a single value. This value is specified as the last instruction in the statement expression. In this case, the value is the result of the asm statement, my_stack_ptr. By writing an asm macro with a statement expression, the asm result may be assigned directly to another variable (for example, void * stack_pointer = GET_SP2) or included in a larger expression, such as: void * stack_pointer = GET_SP2 - sizeof(long).

Which style of macro to use depends on the application. If the asm statement needs to be a part of an expression, then a macro with a statement expression is a good approach. Otherwise, a traditional macro, like GET_SP(x), will probably suffice.

Intrinsics

Inline intrinsic functions map to actual x86 or x64 machine instructions. Intrinsics are inserted inline to avoid the overhead of a function call. The compiler has special knowledge of intrinsics, so with use of intrinsics, better code may be generated as compared to extended inline assembly code.

The PGI Workstation version 7.0 or higher compiler intrinsics library implements MMX, SSE, SS2, SSE3, SSSE3, SSE4a, and ABM instructions. The intrinsic functions are available to C and C++ programs on Linux and Windows. Unlike most functions which are in libraries, intrinsics are implemented internally by the compiler. A program can call the intrinsic functions from C/C++ source code after including the corresponding header file.

The intrinsics are divided into header files as follows:

Table 14.6. Intrinsic Header File Organization

Instructions	Header File
MMX	mmintrin.h
SSE	xmmintrin.h
SSE2	emmintrin.h
SSE3	pmmintrin.h
SSSE3	tmmintrin.h
SSE4a	ammintrin.h
ABM	intrin.h

The following is a simple example program that calls XMM intrinsics.

```
#include <xmmintrin.h>
int main(){
 __m128, __A, __B, result;
 __A = _mm_set_ps(23.3, 43.7, 234.234, 98.746);
 __B = _mm_set_ps(15.4, 34.3, 4.1, 8.6);
 result = _mm_add_ps(__A,__B);
 return 0;
}
```

Part II. Reference Information

In Part I you learned how to use the PGI compilers as well as why certain options or tasks are useful in enhancing the effectiveness and efficiency of the PGI compilers and tools. You may now be ready to learn more about specific areas or specific topics. The chapters in this part of the guide provide more data and facts about the topics that you have already learned about, including information about:

- Data types, as described in Chapter 15, "Fortran, C and C++ Data Types" on page 187.
- Detailed information about each of the command-line options, as described in Chapter 16, "Command-Line Options Reference" on page 195.
- Details about the OpenMP directives and pragmas, as described in Chapter 17, "OpenMP Reference Information" on page 283.
- PGI Accelerator directives, runtime routines, and environment variables, as described in Chapter 18, "PGI Accelerator Compilers Reference" on page 311.
- C++ Name Mangling, as described in Chapter 19, "C++ Name Mangling" on page 323.
- Details about PGI directives and pragmas, as described in Chapter 20, "Directives and Pragmas Reference" on page 327.
- Information about run-time environments, as described in Chapter 21, "Run-time Environment" on page 339.
- C++ dialect that are supported, as described in Chapter 22, "C++ Dialect Supported" on page 367.
- Fortran module and library interfaces that PGI uses to support the Win32 API and Unix/Linux/Mac OS X portability libraries, as described in Chapter 23, Fortran Module/Library Interfaces for Windows" on page 371.
- C and C++ Inline Intrinsics, as described in Chapter 24, "C/C++ MMX/SSE Inline Intrinsics" on page 401.
- Error messages, as described in Chapter 25, "Messages" on page 409.

Chapter 15. Fortran, C, and C++ Data Types

This chapter describes the scalar and aggregate data types recognized by the PGI Fortran, C, and C++ compilers, the format and alignment of each type in memory, and the range of values each type can have on x86 or x64 processor-based systems running a 32-bit operating system. For more information on x86-specific data representation, refer to the System V Application Binary Interface, Processor Supplement, listed in "Related Publications," on page xxvi. This chapter specifically does not address x64 processor-based systems running a 64-bit operating system, because the application binary interface (ABI) for those systems is still evolving. For the latest version of the ABI, refer to *www.x86-64.org/abi.pdf*.

Fortran Data Types

Fortran Scalars

A scalar data type holds a single value, such as the integer value 42 or the real value 112.6. The next table lists scalar data types, their size, format and range. Table 15.2, "Real Data Type Ranges," on page 188 shows the range and approximate precision for Fortran real data types. Table 15.3, "Scalar Type Alignment," on page 188 shows the alignment for different scalar data types. The alignments apply to all scalars, whether they are independent or contained in an array, a structure or a union.

Table 15.1. Representation of Fortran Data Types

Fortran Data Type	Format	Range
INTEGER	2's complement integer	-2^{31} to $2^{31}-1$
INTEGER*2	2's complement integer	-32768 to 32767
INTEGER*4	2's complement integer	-2^{31} to $2^{31}-1$
INTEGER*8	2's complement integer	-2^{63} to $2^{63}-1$
LOGICAL	32-bit value	true or false
LOGICAL*1	8-bit value	true or false
LOGICAL*2	16-bit value	true or false

Fortran Data Type	Format	Range
LOGICAL*4	32-bit value	true or false
LOGICAL*8	64-bit value	true or false
BYTE	2's complement	-128 to 127
REAL	Single-precision floating point	10^{-37} to $10^{38(1)}$
REAL*4	Single-precision floating point	10^{-37} to $10^{38(1)}$
REAL*8	Double-precision floating point	10^{-307} to $10^{308(1)}$
DOUBLE PRECISION	Double-precision floating point	10^{-307} to $10^{308\ (1)}$
COMPLEX	Single-precision floating point	10^{-37} to $10^{38\ (1)}$
DOUBLE COMPLEX	Double-precision floating point	10^{-307} to $10^{308\ (1)}$
COMPLEX*16	Double-precision floating point	10^{-307} to $10^{308(1)}$
CHARACTER*n	Sequence of n bytes	

(1) Approximate value

The logical constants `.TRUE.` and `.FALSE.` are all ones and all zeroes, respectively. Internally, the value of a logical variable is true if the least significant bit is one and false otherwise. When the option `-Munixlogical` is set, a logical variable with a non-zero value is true and with a zero value is false.

Note

A variable of logical type may appear in an arithmetic context, and the logical type is then treated as an integer of the same size.

Table 15.2. Real Data Type Ranges

Data Type	Binary Range	Decimal Range	Digits of Precision
REAL	-2^{-126} to 2^{128}	10^{-37} to $10^{38\ (1)}$	7-8
REAL*8	-2^{-1022} to 2^{1024}	10^{-307} to $10^{308\ (1)}$	15-16

Table 15.3. Scalar Type Alignment

This Type...	...Is aligned on this size boundary
LOGICAL*1	1-byte
LOGICAL*2	2-byte
LOGICAL*4	4-byte
LOGICAL*8	8-byte
BYTE	1-byte
INTEGER*2	2-byte
INTEGER*4	4-byte

This Type...	...Is aligned on this size boundary
INTEGER*8	8-byte
REAL*4	4-byte
REAL*8	8-byte
COMPLEX*8	4-byte
COMPLEX*16	8-byte

FORTRAN 77 Aggregate Data Type Extensions

The PGF77 compiler supports de facto standard extensions to FORTRAN 77 that allow for aggregate data types. An aggregate data type consists of one or more scalar data type objects. You can declare the following aggregate data types:

- An **array** consists of one or more elements of a single data type placed in contiguous locations from first to last.
- A **structure** can contain different data types. The members are allocated in the order they appear in the definition but may not occupy contiguous locations.
- A **union** is a single location that can contain any of a specified set of scalar or aggregate data types. A union can have only one value at a time. The data type of the union member to which data is assigned determines the data type of the union after that assignment.

The alignment of an array, a structure or union (an aggregate) affects how much space the object occupies and how efficiently the processor can address members. Arrays use the alignment of their members.

Array types
: align according to the alignment of the array elements. For example, an array of INTEGER*2 data aligns on a 2byte boundary.

Structures and Unions
: align according to the alignment of the most restricted data type of the structure or union. In the next example, the union aligns on a 4byte boundary since the alignment of c, the most restrictive element, is four.

```
STRUCTURE /astr/
UNION
 MAP
 INTEGER*2 a ! 2 bytes
 END MAP
 MAP
 BYTE b ! 1 byte
 END MAP
 MAP
 INTEGER*4 c ! 4 bytes
 END MAP
END UNION
END STRUCTURE
```

Structure alignment can result in unused space called *padding*. Padding between members of the structure is called *internal padding*. Padding between the last member and the end of the space is called *tail padding*.

The offset of a structure member from the beginning of the structure is a multiple of the member's alignment. For example, since an INTEGER*2 aligns on a 2-byte boundary, the offset of an INTEGER*2 member from the beginning of a structure is a multiple of two bytes.

Fortran 90 Aggregate Data Types (Derived Types)

The Fortran 90 standard added formal support for aggregate data types. The TYPE statement begins a derived type data specification or declares variables of a specified user-defined type. For example, the following would define a derived type ATTENDEE:

```
TYPE ATTENDEE
 CHARACTER(LEN=30) NAME
 CHARACTER(LEN=30) ORGANIZATION
 CHARACTER (LEN=30) EMAIL
END TYPE ATTENDEE
```

In order to declare a variable of type ATTENDEE and access the contents of such a variable, code such as the following would be used:

```
TYPE (ATTENDEE) ATTLIST(100)
. . .
ATTLIST(1)%NAME = 'JOHN DOE'
```

C and C++ Data Types

C and C++ Scalars

Table 15.4, "C/C++ Scalar Data Types"lists C and C++ scalar data types, providing their size and format. The alignment of a scalar data type is equal to its size. Table 15.5, "Scalar Alignment," on page 191 shows scalar alignments that apply to individual scalars and to scalars that are elements of an array or members of a structure or union. Wide characters are supported (character constants prefixed with an L). The size of each wide character is 4 bytes.

Table 15.4. C/C++ Scalar Data Types

Data Type	Size (bytes)	Format	Range
unsigned char	1	ordinal	0 to 255
[signed] char	1	2's complement integer	-128 to 127
unsigned short	2	ordinal	0 to 65535
[signed] short	2	2's complement integer	-32768 to 32767
unsigned int	4	ordinal	0 to 2^{32} -1
[signed] int	4	2's complement integer	-2^{31} to 2^{31}-1
[signed] long [int] (32-bit operating systems and win64)	4	2's complement integer	-2^{31} to 2^{31}-1
[signed] long [int] (linux86-64 and sua64)	8	2's complement integer	-2^{63} to 2^{63}-1

Data Type	Size (bytes)	Format	Range
unsigned long [int] (32-bit operating systems and win64)	4	ordinal	0 to 2^{32}-1
unsigned long [int] (linux86-64 and sua64)	8	ordinal	0 to 2^{64}-1
[signed] long long [int]	8	2's complement integer	-2^{63} to 2^{63}-1
unsigned long long [int]	8	ordinal	0 to 2^{64}-1
float	4	IEEE single-precision floating-point	10^{-37} to 10^{38} (1)
double	8	IEEE double-precision floating-point	10^{-307} to 10^{308} (1)
long double	8	IEEE double-precision floating-point	10^{-307} to 10^{308} (1)
bit field(2) (unsigned value)	1 to 32 bits	ordinal	0 to 2^{size}-1, where size is the number of bits in the bit field
bit field(2) (signed value)	1 to 32 bits	2's complement integer	-2^{size-1} to 2^{size-1}-1, where size is the number of bits in the bit field
pointer	4	address	0 to 2^{32}-1
enum	4	2's complement integer	-2^{31} to 2^{31}-1

(1) Approximate value

(2) Bit fields occupy as many bits as you assign them, up to 4 bytes, and their length need not be a multiple of 8 bits (1 byte)

Table 15.5. Scalar Alignment

Data Type	Alignment on this size boundary
char	1-byte boundary, signed or unsigned.
short	2-byte boundary, signed or unsigned.
int	4-byte boundary, signed or unsigned.
enum	4-byte boundary.
pointer	4-byte boundary.
float	4-byte boundary.
double	8-byte boundary.
long double	8-byte boundary.
long [int] 32-bit on Win64	4-byte boundary, signed or unsigned.
long [int] linux86-64, sua64	8-byte boundary, signed or unsigned.

Data Type	Alignment on this size boundary
long long [int]	8-byte boundary, signed or unsigned.

C and C++ Aggregate Data Types

An aggregate data type consists of one or more scalar data type objects. You can declare the following aggregate data types:

array
: consists of one or more elements of a single data type placed in contiguous locations from first to last.

class
: (C++ only) is a class that defines an object and its member functions. The object can contain fundamental data types or other aggregates including other classes. The class members are allocated in the order they appear in the definition but may not occupy contiguous locations.

struct
: is a structure that can contain different data types. The members are allocated in the order they appear in the definition but may not occupy contiguous locations. When a struct is defined with member functions, its alignment rules are the same as those for a class.

union
: is a single location that can contain any of a specified set of scalar or aggregate data types. A union can have only one value at a time. The data type of the union member to which data is assigned determines the data type of the union after that assignment.

Class and Object Data Layout

Class and structure objects with no virtual entities and with no base classes, that is just direct data field members, are laid out in the same manner as C structures. The following section describes the alignment and size of these C-like structures. C++ classes (and structures as a special case of a class) are more difficult to describe. Their alignment and size is determined by compiler generated fields in addition to user-specified fields. The following paragraphs describe how storage is laid out for more general classes. The user is warned that the alignment and size of a class (or structure) is dependent on the existence and placement of direct and virtual base classes and of virtual function information. The information that follows is for informational purposes only, reflects the current implementation, and is subject to change. Do not make assumptions about the layout of complex classes or structures.

All classes are laid out in the same general way, using the following pattern (in the sequence indicated):

- First, storage for all of the direct base classes (which implicitly includes storage for non-virtual indirect base classes as well):
  - When the direct base class is also virtual, only enough space is set aside for a pointer to the actual storage, which appears later.
  - In the case of a non-virtual direct base class, enough storage is set aside for its own non-virtual base classes, its virtual base class pointers, its own fields, and its virtual function information, but no space is allocated for its virtual base classes.

- Next, storage for the class's own fields.
- Next, storage for virtual function information (typically, a pointer to a virtual function table).
- Finally, storage for its virtual base classes, with space enough in each case for its own non-virtual base classes, virtual base class pointers, fields, and virtual function information.

Aggregate Alignment

The alignment of an array, a structure or union (an aggregate) affects how much space the object occupies and how efficiently the processor can address members.

Arrays
: align according to the alignment of the array elements. For example, an array of short data type aligns on a 2-byte boundary.

Structures and Unions
: align according to the most restrictive alignment of the enclosing members. In the following example, the union un1 aligns on a 4-byte boundary since the alignment of c, the most restrictive element, is four:

```
union un1 {
 short a; /* 2 bytes */
 char b; /* 1 byte */
 int c; /* 4 bytes */
};
```

Structure alignment can result in unused space, called padding. Padding between members of a structure is called internal padding. Padding between the last member and the end of the space occupied by the structure is called tail padding. Figure 15.1, "Internal Padding in a Structure," on page 193, illustrates structure alignment. Consider the following structure:

```
struct strc1 {
 char a; /* occupies byte 0 */
 short b; /* occupies bytes 2 and 3 */
 char c; /* occupies byte 4 */
 int d; /* occupies bytes 8 through 11 */
};
```

Figure 15.1. Internal Padding in a Structure

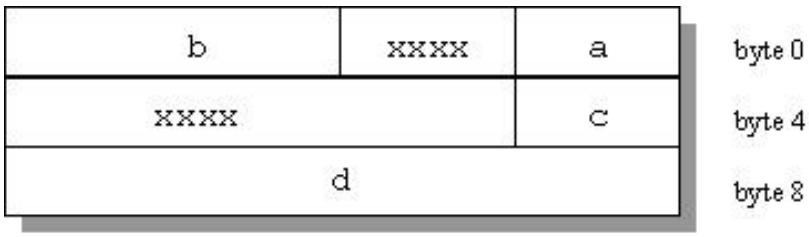


Figure 15.2, "Tail Padding in a Structure," on page 194, shows how tail padding is applied to a structure aligned on a doubleword (8 byte) boundary.

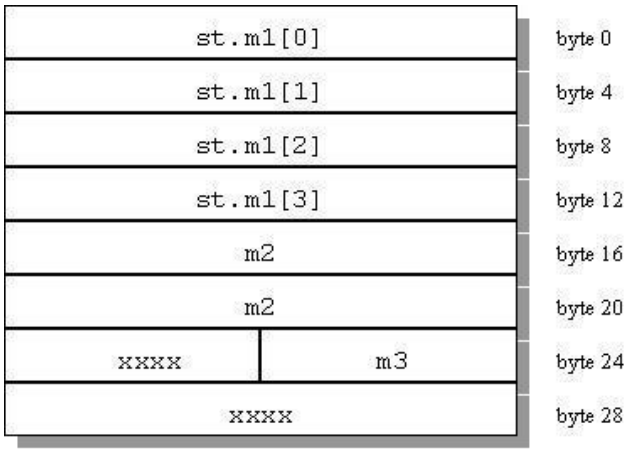
```
struct strc2{
 int m1[4]; /* occupies bytes
0 through 15 */
 double m2; /* occupies bytes 16 through 23 */
 short m3; /* occupies bytes 24 and 25 */
} st;
```

Bit-field Alignment

Bit-fields have the same size and alignment rules as other aggregates, with several additions to these rules:

- Bit-fields are allocated from right to left.
- A bit-field must entirely reside in a storage unit appropriate for its type. Bit-fields never cross unit boundaries.
- Bit-fields may share a storage unit with other structure/union members, including members that are not bit-fields.
- Unnamed bit-field's types do not affect the alignment of a structure or union.
- Items of [signed/unsigned] long long type may not appear in field declarations on 32-bit systems.

Figure 15.2. Tail Padding in a Structure



Other Type Keywords in C and C++

The void data type is neither a scalar nor an aggregate. You can use void or void* as the return type of a function to indicate the function does not return a value, or as a pointer to an unspecified data type, respectively.

The const and volatile type qualifiers do not in themselves define data types, but associate attributes with other types. Use const to specify that an identifier is a constant and is not to be changed. Use volatile to prevent optimization problems with data that can be changed from outside the program, such as memory#mapped I/O buffers.

Chapter 16. Command-Line Options Reference

A command-line option allows you to specify specific behavior when a program is compiled and linked. Compiler options perform a variety of functions, such as setting compiler characteristics, describing the object code to be produced, controlling the diagnostic messages emitted, and performing some preprocessor functions. Most options that are not explicitly set take the default settings. This reference chapter describes the syntax and operation of each compiler option. For easy reference, the options are arranged in alphabetical order.

For an overview and tips on which options are best for which tasks, refer to Chapter 2, "*Using Command Line Options*," on page 17, which also provides summary tables of the different options.

This chapter uses the following notation:

[item]
: Square brackets indicate that the enclosed item is optional.

{item | item}
: Braces indicate that you must select one and only one of the enclosed items. A vertical bar (|) separates the choices.

...
: Horizontal ellipses indicate that zero or more instances of the preceding item are valid.

PGI Compiler Option Summary

The following tables include all the PGI compiler options that are not language-specific. The options are separated by category for easier reference.

For a complete description of each option, see the detailed information later in this chapter.

Build-Related PGI Options

The options included in the following table are the ones you use when you are initially building your program or application.

Table 16.1. PGI Build-Related Compiler Options

Option	Description
–#	Display invocation information.
–###	Shows but does not execute the driver commands (same as –dryrun).
–Bdynamic	Compiles for and links to the DLL version of the PGI runtime libraries.
–Bstatic_pgi	Compiles for and links to the static version of the PGI runtime libraries.
–c	Stops after the assembly phase and saves the object code in filename.o.
–D<args>	Defines a preprocessor macro.
–d<arg>	Prints additional information from the preprocessor.
–dryrun	Shows but does not execute driver commands.
–drystdinc	Displays the standard include directories and then exists the compiler.
–dynamiclib	Invokes the libtool utility program provided by Mac OS X to create the dynamic library. See the libtool man page for more information.
–E	Stops after the preprocessing phase and displays the preprocessed file on the standard output.
–F	Stops after the preprocessing phase and saves the preprocessed file in filename.f (this option is only valid for the PGI Fortran compilers).
--flagcheck	Simply return zero status if flags are correct.
–flags	Display valid driver options.
–fpic	(Linux and Mac OS X only) Generate position-independent code.
–fPIC	(Linux and Mac OS X only) Equivalent to –fpic.
–G	(Linux only) Passed to the linker. Instructs the linker to produce a shared object file.
–g77libs	(Linux only) Allow object files generated by g77 to be linked into PGI main programs.
–help	Display driver help message.
–I <dirname>	Adds a directory to the search path for #include files.
–i2, –i4 and –i8	–i2: Treat INTEGER variables as 2 bytes. –i4: Treat INTEGER variables as 4 bytes. –i8: Treat INTEGER and LOGICAL variables as 8 bytes and use 64-bits for INTEGER*8 operations.

Option	Description
–K<flag>	Requests special compilation semantics with regard to conformance to IEEE 754.
--keeplnk	If the compiler generates a temporary indirect file for a long linker command, preserves the temporary file instead of deleting it.
–L<dirname>	Specifies a library directory.
–l<library>	Loads a library.
–m	Displays a link map on the standard output.
–M<pgflag>	Selects variations for code generation and optimization.
–mcmodel=medium	(–tp k8-64 and –tp p7-64 targets only) Generate code which supports the medium memory model in the linux86-64 environment.
–module <moduledir>	(F90/F95/HPF only) Save/search for module files in directory <moduledir>.
–mp[=all, align,bind, [no]numa]	Interpret and process user-inserted shared-memory parallel programming directives (see Chapters 5 and 6).
–noswitcherror	Ignore unknown command line switches after printing an warning message.
–o	Names the object file.
–pc <val>	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
--pedantic	Prints warnings from included <system header files>
–pg	Instrument the generated executable to produce a gprof-style gmon.out sample-based profiling trace file (–qp is also supported, and is equivalent).
–pgf77libs	Append PGF77 runtime libraries to the link line.
–pgf90libs	Append PGF90/PGF95 runtime libraries to the link line.
–R<directory>	(Linux only) Passed to the Linker. Hard code <directory> into the search path for shared object files.
–r	Creates a relocatable object file.
–r4 and –r8	–r4: Interpret DOUBLE PRECISION variables as REAL. –r8: Interpret REAL variables as DOUBLE PRECISION.
–rc file	Specifies the name of the driver's startup file.
–s	Strips the symbol-table information from the object file.
–S	Stops after the compiling phase and saves the assembly–language code in filename.s.

Option	Description
–shared	(Linux only) Passed to the linker. Instructs the linker to generate a shared object file. Implies –fpic.
–show	Display driver's configuration parameters after startup.
–silent	Do not print warning messages.
–soname	Pass the soname option and its argument to the linker.
–time	Print execution times for the various compilation steps.
–tp <target> [,target...]	Specify the type(s) of the target processor(s).
–u <symbol>	Initializes the symbol table with <symbol>, which is undefined for the linker. An undefined symbol triggers loading of the first member of an archive library.
–U <symbol>	Undefine a preprocessor macro.
–V[release_number]	Displays the version messages and other information, or allows invocation of a version of the compiler other than the default.
–v	Displays the compiler, assembler, and linker phase invocations.
–W	Passes arguments to a specific phase.
–w	Do not print warning messages.

PGI Debug-Related Compiler Options

The options included in the following table are the ones you typically use when you are debugging your program or application.

Table 16.2. PGI Debug-Related Compiler Options

Option	Description
–C	(Fortran only) Generates code to check array bounds.
–c	Instrument the generated executable to perform array bounds checking at runtime.
–E	Stops after the preprocessing phase and displays the preprocessed file on the standard output.
--flagcheck	Simply return zero status if flags are correct.
–flags	Display valid driver options.
–g	Includes debugging information in the object module.
–gopt	Includes debugging information in the object module, but forces assembly code generation identical to that obtained when `-gopt` is not present on the command line.
–K<flag>	Requests special compilation semantics with regard to conformance to IEEE 754.

Option	Description
--keeplnk	If the compiler generates a temporary indirect file for a long linker command, preserves the temporary file instead of deleting it.
–M<pgflag>	Selects variations for code generation and optimization.
–pc <val>	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
–Mprof=time	Instrument the generated executable to produce a gprof-style gmon.out sample-based profiling trace file (–qp is also supported, and is equivalent).

PGI Optimization-Related Compiler Options

The options included in the following table are the ones you typically use when you are optimizing your program or application code.

Table 16.3. Optimization-Related PGI Compiler Options

Option	Description
–fast	Generally optimal set of flags for targets that support SSE capability.
–fastsse	Generally optimal set of flags for targets that include SSE/SSE2 capability.
–M<pgflag>	Selects variations for code generation and optimization.
–mp[=all, align,bind, [no]numa]	Interpret and process user-inserted shared-memory parallel programming directives (see Chapters 5 and 6).
–O<level>	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
–pc <val>	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
–Mprof=time	Instrument the generated executable to produce a gprof-style gmon.out sample-based profiling trace file (-qp is also supported, and is equivalent).

PGI Linking and Runtime-Related Compiler Options

The options included in the following table are the ones you typically use to define parameters related to linking and running your program or application code.

Table 16.4. Linking and Runtime-Related PGI Compiler Options

Option	Description
–Bdynamic	Compiles for and links to the DLL version of the PGI runtime libraries.

Option	Description
–Bstatic_pgi	Compiles for and links to the static version of the PGI runtime libraries.
–byteswapio	(Fortran only) Swap bytes from big-endian to little-endian or vice versa on input/output of unformatted data
–fpic	(Linux only) Generate position-independent code.
–fPIC	(Linux only) Equivalent to –fpic.
–G	(Linux only) Passed to the linker. Instructs the linker to produce a shared object file.
–g77libs	(Linux only) Allow object files generated by g77 to be linked into PGI main programs.
–i2	Treat INTEGER variables as 2 bytes.
–i4	Treat INTEGER variables as 4 bytes.
–i8	Treat INTEGER and LOGICAL variables as 8 bytes and use 64-bits for INTEGER*8 operations.
–K<flag>	Requests special compilation semantics with regard to conformance to IEEE 754.
–M<pgflag>	Selects variations for code generation and optimization.
–mcmodel=medium	(–tp k8-64 and –tp p7-64 targets only) Generate code which supports the medium memory model in the linux86-64 environment.
–shared	(Linux only) Passed to the linker. Instructs the linker to generate a shared object file. Implies –fpic.
–soname	Pass the soname option and its argument to the linker.
–ta=nvidia(,nvidia_suboptions),host	Specify the target accelerator.
–tp <target> [,target...]	Specify the type(s) of the target processor(s).

C and C++ Compiler Options

There are a large number of compiler options specific to the PGCC and PGC++ compilers, especially PGC++. The next table lists several of these options, but is not exhaustive. For a complete list of available options, including an exhaustive list of PGC++ options, use the –help command-line option. For further detail on a given option, use –help and specify the option explicitly. The majority of these options are related to building your program or application.

Table 16.5. C and C++ -specific Compiler Options

Option	Description
–A	(pgcpp only) Accept proposed ANSI C++, issuing errors for non-conforming code.

Option	Description
–a	(pgcpp only) Accept proposed ANSI C++, issuing warnings for non-conforming code.
--[no_]alternative_tokens	(pgcpp only) Enable/disable recognition of alternative tokens. These are tokens that make it possible to write C++ without the use of the , , `[, ]`, #, &, and ^ and characters. The alternative tokens include the operator keywords (e.g., and, bitand, etc.) and digraphs. The default is -–no_alternative_tokens.
–B	Allow C++ comments (using //) in C source.
–b	(pgcpp only) Compile with cfront 2.1 compatibility. This accepts constructs and a version of C++ that is not part of the language definition but is accepted by cfront. EDG option.
–b3	(pgcpp only) Compile with cfront 3.0 compatibility. See –b.
--[no_]bool	(pgcpp only) Enable or disable recognition of bool. The default value is —-bool.
--[no_]builtin	Do/don't compile with math subroutine builtin support, which causes selected math library routines to be inlined. The default is —-builtin.
--cfront_2.1	(pgcpp only) Enable compilation of C++ with compatibility with cfront version 2.1.
--cfront_3.0	(pgcpp only) Enable compilation of C++ with compatibility with cfront version 3.0.
--compress_names	(pgcpp only) Create a precompiled header file with the name filename.
--dependencies (see –M)	(pgcpp only) Print makefile dependencies to stdout.
--dependencies_to_file `filename`	(pgcpp only) Print makefile dependencies to file `filename`.
--display_error_number	(pgcpp only) Display the error message number in any diagnostic messages that are generated.
--diag_error tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.
--diag_remark tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.
--diag_suppress tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.
--diag_warning tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.

Option	Description
-e<number>	(pgcpp only) Set the C++ front-end error limit to the specified <number>.
--[no_]exceptions	(pgcpp only) Disable/enable exception handling support. The default is —-exceptions
--gnu_extensions	(pgcpp only) Allow GNU extensions like "include next" which are required to compile Linux system header files.
--[no]llalign	(pgcpp only) Do/don't align longlong integers on integer boundaries. The default is —-llalign.
–M	Generate make dependence lists.
–MD	Generate make dependence lists.
–MD,filename	(pgcpp only) Generate make dependence lists and print them to file filename.
--optk_allow_dollar_in_id_chars	(pgcpp only) Accept dollar signs in identifiers.
–P	Stops after the preprocessing phase and saves the preprocessed file in filename.i.
--+p	(pgcpp only) Disallow all anachronistic constructs. cfront option
--pch	(pgcpp only) Automatically use and/or create a precompiled header file.
--pch_dir directoryname	(pgcpp only) The directory dirname in which to search for and/or create a precompiled header file.
--[no_]pch_messages	(pgcpp only) Enable/ disable the display of a message indicating that a precompiled header file was created or used.
--preinclude=<filename>	(pgcpp only) Specify file to be included at the beginning of compilation so you can set system-dependent macros, types, and so on.
–t	Control instantiation of template functions. EDG option
--use_pch filename	(pgcpp only) Use a precompiled header file of the specified name as part of the current compilation.
--[no_]using_std	(pgcpp only) Enable/disable implicit use of the std namespace when standard header files are included.
–X	(pgcpp only) Allow $ in names.

Generic PGI Compiler Options

The following descriptions are for all the PGI options. For easy reference, the options are arranged in alphabetical order. For a list of options by tasks, refer to the tables in the beginning of this chapter as well as to Chapter 2, "*Using Command Line Options*," on page 17.

–#

Displays the invocations of the compiler, assembler and linker.

Default: The compiler does not display individual phase invocations.

Usage: The following command-line requests verbose invocation information.

```
$ pgf95 -# prog.f
```

Description: The –# option displays the invocations of the compiler, assembler and linker. These invocations are command-lines created by the driver from your command-line input and the default value.

Related options: –Minfo, –V, –v.

–###

Displays the invocations of the compiler, assembler and linker, but does not execute them.

Default: The compiler does not display individual phase invocations.

Usage: The following command-line requests verbose invocation information.

```
$ pgf95 -### myprog.f
```

Description: Use the –### option to display the invocations of the compiler, assembler and linker but not to execute them. These invocations are command lines created by the compiler driver from the `rc` files and the command-line options.

Related options: –#, –dryrun, –Minfo, –V

–Bdynamic

Compiles for and links to the DLL version of the PGI runtime libraries.

Default: The compiler uses static libraries.

Usage: You can create the DLL `obj1.dll` and its import library `obj1.lib` using the following series of commands:

```
% pgf95 -Bdynamic -c object1.f
% pgf95 -Mmakedll object1.obj -o obj1.dll
```

Then compile the main program using this command:

```
$ pgf95 -# prog.f
```

For a complete example, refer to Example 9.1, "Build a DLL: Fortran," on page 119.

Description: Use this option to compile for and link to the DLL version of the PGI runtime libraries. This flag is required when linking with any DLL built by the PGI compilers. This flag corresponds to the `/MD` flag used by Microsoft's cl compilers.

Note

On Windows, `-Bdynamic` must be used for *both* compiling and linking.

When you use the PGI compiler flag `-Bdynamic` to create an executable that links to the DLL form of the runtime, the executable built is smaller than one built without `-Bdynamic`. The PGI runtime DLLs, however, must be available on the system where the executable is run. The `-Bdynamic` flag must be used when an executable is linked against a DLL built by the PGI compilers.

Related options:–Bstatic, –Mmakedll

–Bstatic

Compiles for and links to the static version of the PGI runtime libraries.

Default: The compiler uses static libraries.

Usage: The following command line explicitly compiles for and links to the static version of the PGI runtime libraries:

```
% pgf95 -Bstatic -c object1.f
```

Description: You can use this option to explicitly compile for and link to the static version of the PGI runtime libraries.

Note

On Windows, `-Bstatic` must be used for *both* compiling and linking.

For more information on using static libraries on Windows, refer to "Creating and Using Static Libraries on Windows," on page 116.

Related options: –Bdynamic, –Bstatic_pgi, –Mdll

–Bstatic_pgi

Linux only. Compiles for and links to the static version of the PGI runtime libraries. Implies `-Mnorpath`.

Default: The compiler uses static libraries.

Usage: The following command line explicitly compiles for and links to the static version of the PGI runtime libraries:

```
% pgf95 -Bstatic -c object1.f
```

Description: You can use this option to explicitly compile for and link to the static version of the PGI runtime libraries.

Note

On Linux, `-Bstatic_pgi` results in code that runs on most Linux systems without requiring a Portability package.

For more information on using static libraries on Linux, refer to "Creating and Using Static Libraries on Windows," on page 116.

Related options: –Bdynamic, –Bstatic, –Mdll

–byteswapio

Swaps the byte-order of data in unformatted Fortran data files on input/output.

Default: The compiler does not byte-swap data on input/output.

Usage: The following command-line requests that byte-swapping be performed on input/output.

```
$ pgf95 -byteswapio myprog.f
```

Description: Use the `-byteswapio` option to swap the byte-order of data in unformatted Fortran data files on input/output. When this option is used, the order of bytes is swapped in both the data and record control words; the latter occurs in unformatted sequential files.

You can use this option to convert big-endian format data files produced by most RISC workstations and high-end servers to the little-endian format used on x86 or x64 systems on the fly during file reads/writes.

This option assumes that the record layouts of unformatted sequential access and direct access files are the same on the systems. It further assumes that the IEEE representation is used for floating-point numbers. In particular, the format of unformatted data files produced by PGI Fortran compilers is identical to the format used on Sun and SGI workstations; this format allows you to read and write unformatted Fortran data files produced on those platforms from a program compiled for an x86 or x64 platform using the `-byteswapio` option.

Related options: None.

–C

Enables array bounds checking. This option only applies to the PGI Fortran compilers.

Default: The compiler does not enable array bounds checking.

Usage: In this example, the compiler instruments the executable produced from `myprog.f` to perform array bounds checking at runtime:

```
$ pgf95 -C myprog.f
```

Description: Use this option to enable array bounds checking. If an array is an assumed size array, the bounds checking only applies to the lower bound. If an array bounds violation occurs during execution, an error message describing the error is printed and the program terminates. The text of the error message includes the name of the array, the location where the error occurred (the source file and the line number in

the source), and information about the out of bounds subscript (its value, its lower and upper bounds, and its dimension).

Related options: –M[no]bounds.

–c

Halts the compilation process after the assembling phase and writes the object code to a file.

Default: The compiler produces an executable file (does not use the –c option).

Usage: In this example, the compiler produces the object file `myprog.o` in the current directory.

```
$ pgf95 -c myprog.f
```

Description: Use the –c option to halt the compilation process after the assembling phase and write the object code to a file. If the input file is `filename.f`, the output file is `filename.o`.

Related options: –E, –Mkeepasm, –o, and –S.

–d<arg>

Prints additional information from the preprocessor. Valid only for C (pgcc).

Default: No additional information is printed from the preprocessor.

Syntax:

```
-d[D|I|M|N]
```

-dD
: Print macros and values from source files.

-dI
: Print include file names.

-dM
: Print macros and values, including predefined and command-line macros.

-dN
: Print macro names from source files.

Usage: In the following example, the compiler prints macro names from the source file.

```
$ pgf95 -dN myprog.f
```

Description: Use the -d<arg> option to print additional information from the preprocessor.

Related options: –E, –D, –U.

–D

Creates a preprocessor macro with a given value.

Note

You can use the –D option more than once on a compiler command line. The number of active macro definitions is limited only by available memory.

Syntax:

```
-Dname[=value]
```

Where name is the symbolic name and value is either an integer value or a character string.

Default: If you define a macro name without specifying a value, the preprocessor assigns the string 1 to the macro name.

Usage: In the following example, the macro PATHLENGTH has the value 256 until a subsequent compilation. If the –D option is not used, PATHLENGTH is set to 128.

```
$ pgf95 -DPATHLENGTH=256 myprog.F
```

The source text in `myprog.F` is this:

```
#ifndef PATHLENGTH
#define PATHLENGTH 128
#endif
SUBROUTINE SUB
CHARACTER*PATHLENGTH path
...
END
```

Description: Use the –D option to create a preprocessor macro with a given value. The value must be either an integer or a character string.

You can use macros with conditional compilation to select source text during preprocessing. A macro defined in the compiler invocation remains in effect for each module on the command line, unless you remove the macro with an #undef preprocessor directive or with the –U option. The compiler processes all of the –U options in a command line after processing the –D options.

Related options: –U

–dryrun

Displays the invocations of the compiler, assembler, and linker but does not execute them.

Default: The compiler does not display individual phase invocations.

Usage: The following command line requests verbose invocation information.

```
$ pgf95 -dryrun myprog.f
```

Description: Use the `-dryrun` option to display the invocations of the compiler, assembler, and linker but not have them executed. These invocations are command lines created by the compiler driver from the `rc` files and the command-line supplied with `-dryrun`.

Related options: –Minfo, –V, –###

–drystdinc

Displays the standard include directories and then exits the compiler.

Default: The compiler does not display standard include directores.

Usage: The following command line requests a display for the standard include directories.

```
$ pgf95 -drystdinc myprog.f
```

Description: Use the `-drystdinc` option to display the standard include directories and then exit the compiler.

Related options: None.

–dynamiclib

Invokes the `libtool` utility program provided by Mac OS X to so you can create a dynamic library.

Default: The compiler does not invoke the libtool utility.

Usage: The following command line builds a dynamic library:

```
% pgf95 -dynamiclib world.f90 -o world.dylib
```

Description: Use the –dynamiclib option to invoke the `libtool` utility program provided by Mac OS X to so you can create a dynamic library. For a complete example, refer to "Creating and Using Dynamic Libraries on Mac OS X," on page 115.

For more information on `libtool`, refer to the `libtool` man page.

Related options: –Bdynamic, –Bstatic

–E

Halts the compilation process after the preprocessing phase and displays the preprocessed output on the standard output.

Default: The compiler produces an executable file.

Usage: In the following example the compiler displays the preprocessed `myprog.f` on the standard output.

```
$ pgf95 -E myprog.f
```

Description: Use the –E option to halt the compilation process after the preprocessing phase and display the preprocessed output on the standard output.

Related options: –C, –c, –Mkeepasm, –o, –F, –S.

–F

Stops compilation after the preprocessing phase.

Default: The compiler produces an executable file.

Usage: In the following example the compiler produces the preprocessed file `myprog.f` in the current directory.

```
$ pgf95 -F myprog.F
```

Description: Use the –F option to halt the compilation process after preprocessing and write the preprocessed output to a file. If the input file is `filename.F`, then the output file is `filename.f`.

Related options: –c,–E, –Mkeepasm, –o, –S

–fast

Enables vectorization with SSE instructions, cache alignment, and flushz for 64-bit targets.

Default: The compiler enables vectorization with SSE instructions, cache alignment, and flushz.

Usage: In the following example the compiler produces vector SSE code when targeting a 64-bit machine.

```
$ pgf95 -fast vadd.f95
```

Description: When you use this option, a generally optimal set of options is chosen for targets that support SSE capability. In addition, the appropriate `-tp` option is automatically included to enable generation of code optimized for the type of system on which compilation is performed. This option enables vectorization with SSE instructions, cache alignment, and flushz.

Note

Auto-selection of the appropriate `-tp` option means that programs built using the `-fastsse` option on a given system are not necessarily backward-compatible with older systems.

Note

C/C++ compilers enable `-Mautoinline` with `-fast`.

Related options: –O, –Munroll, –Mnoframe, –Mscalarsse, –Mvect, –Mcache_align, –tp

–fastsse

Synonymous with –fast.

--flagcheck

Causes the compiler to check that flags are correct then exit without any compilation occuring.

Default: The compiler begins a compile without the additional step to first validate that flags are correct.

Usage: In the following example the compiler checks that flags are correct, and then exits.

```
$ pgf95 --flagcheck myprog.f
```

Description: Use this option to make the compiler check that flags are correct and then exit. If flags are all correct then the compiler returns a zero status. No compilation occurs.

Related options: None

–flags

Displays driver options on the standard output.

Default: The compiler does not display the driver options.

Usage: In the following example the user requests information about the known switches.

```
$ pgf95 -flags
```

Description: Use this option to display driver options on the standard output. When you use this option with –v, in addition to the valid options, the compiler lists options that are recognized and ignored.

Related options: –#, –###, –v

–fpic

(Linux only) Generates position-independent code suitable for inclusion in shared object (dynamically linked library) files.

Default: The compiler does not generate position-independent code.

Usage: In the following example the resulting object file, `myprog.o`, can be used to generate a shared object.

```
$ pgf95 -fpic myprog.f
```

(Linux only) Use the -fpic option to generate position-independent code suitable for inclusion in shared object (dynamically linked library) files.

Related options: –shared, –fPIC, –G, –R

–fPIC

(Linux only) Equivalent to –fpic. Provided for compatibility with other compilers.

–G

(Linux only) Instructs the linker to produce a shared object file.

Default: The compiler does not instruct the linker to produce a shared object file.

Usage: In the following example the linker produces a shared object file.

```
$ pgf95 -G myprog.f
```

Description: (Linux only) Use this option to pass information to the linker that instructs the linker to produce a shared object file.

Related options: –fpic, –shared, –R

–g

Instructs the compiler to include symbolic debugging information in the object module.

Default: The compiler does not put debugging information into the object module.

Usage: In the following example, the object file `myprog.o` contains symbolic debugging information.

```
$ pgf95 -c -g myprog.f
```

Description: Use the –g option to instruct the compiler to include symbolic debugging information in the object module. Debuggers, such as PGDBG, require symbolic debugging information in the object module to display and manipulate program variables and source code.

If you specify the –g option on the command-line, the compiler sets the optimization level to –O0 (zero), unless you specify the –O option. For more information on the interaction between the –g and –O options, see the –O entry. Symbolic debugging may give confusing results if an optimization level other than zero is selected.

Note

Including symbolic debugging information increases the size of the object module.

Related options: –O, –gopt

–gopt

Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when –g is not specified.

Default: The compiler does not put debugging information into the object module.

Usage: In the following example, the object file`myprog.o` contains symbolic debugging information.

```
$ pgf95 -c -gopt myprog.f
```

Description: Using `-g` alters how optimized code is generated in ways that are intended to enable or improve debugging of optimized code. The `-gopt` option instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when –g is not specified.

Related options: –g, –M<pgflag>

–g77libs

(Linux only) Used on the link line, this option instructs the pgf95 driver to search the necessary g77 support libraries to resolve references specific to g77 compiled program units.

Note

The g77 compiler must be installed on the system on which linking occurs in order for this option to function correctly.

Default: The compiler does not search g77 support libraries to resolve references at link time.

Usage: The following command-line requests that g77 support libraries be searched at link time:

```
$ pgf95 -g77libs myprog.f g77_object.o
```

Description: (Linux only) Use the –g77libs option on the link line if you are linking g77-compiled program units into a pgf95-compiled main program using the pgf95 driver. When this option is present, the pgf95 driver searches the necessary g77 support libraries to resolve references specific to g77 compiled program units.

Related options:–pgf77libs

–help

Used with no other options, –help displays options recognized by the driver on the standard output. When used in combination with one or more additional options, usage information for those options is displayed to standard output.

Default: The compiler does not display usage information.

Usage: In the following example, usage information for `-Minline` is printed to standard output.

```
$ pgcc -help -Minline
-Minline[=lib:<inlib>|<func>|except:<func>|
name:<func>|size:<n>|levels:<n>]
Enable function inlining
lib:<extlib> Use extracted functions from extlib
<func> Inline function func
except:<func> Do not inline function func
name:<func> Inline function func
size:<n> Inline only functions smaller than n
levels:<n> Inline n levels of functions
-Minline Inline all functions that were extracted
```

In the following example, usage information for –help shows how groups of options can be listed or examined according to function.

```
$ pgcc -help -help
-help[=groups|asm|debug|language|linker|opt|other|
 overall|phase|prepro|suffix|switch|target|variable]
```

Description: Use the –help option to obtain information about available options and their syntax. You can use –help in one of three ways:

- Use `-help` with no parameters to obtain a list of all the available options with a brief one-line description of each.
- Add a parameter to –help to restrict the output to information about a specific option. The syntax for this usage is this:

  ```
  -help <command line option>
  ```

- Add a parameter to –help to restrict the output to a specific set of options or to a building process. The syntax for this usage is this:

  ```
  -help=<subgroup>
  ```

The following table lists and describes the subgroups available with –help.

Table 16.6. Subgroups for –help Option

Use this –help option	To get this information...
–help=asm	A list of options specific to the assembly phase.
–help=debug	A list of options related to debug information generation.
–help=groups	A list of available switch classifications.
–help=language	A list of language-specific options.
–help=linker	A list of options specific to link phase.
–help=opt	A list of options specific to optimization phase.
–help=other	A list of other options, such as ANSI conformance pointer aliasing for C.
–help=overall	A list of options generic to any PGI compiler.
–help=phase	A list of build process phases and to which compiler they apply.
–help=prepro	A list of options specific to the preprocessing phase.
–help=suffix	A list of known file suffixes and to which phases they apply.
–help=switch	A list of all known options; this is equivalent to usage of –help without any parameter.
–help=target	A list of options specific to target processor.
–help=variable	A list of all variables and their current value. They can be redefined on the command line using syntax VAR=VALUE.

For more examples of –help, refer to "Help with Command-line Options," on page 18.

Related options: –#, –###, –show, –V, –flags

–I

Adds a directory to the search path for files that are included using either the INCLUDE statement or the preprocessor directive #include.

Default: The compiler searches only certain directories for included files.

- For gcc-lib includes: `/usr/lib64/gcc-lib`
- For system includes: `/usr/linclude`

Syntax:

```
-Idirectory
```

Where directory is the name of the directory added to the standard search path for include files.

Usage: In the following example, the compiler first searches the directory `mydir` and then searches the default directories for include files.

```
$ pgf95 -Imydir
```

Description: Adds a directory to the search path for files that are included using the INCLUDE statement or the preprocessor directive #include. Use the –I option to add a directory to the list of where to search for the included files. The compiler searches the directory specified by the –I option before the default directories.

The Fortran INCLUDE statement directs the compiler to begin reading from another file. The compiler uses two rules to locate the file:

1. If the file name specified in the INCLUDE statement includes a path name, the compiler begins reading from the file it specifies.
2. If no path name is provided in the INCLUDE statement, the compiler searches (in order):
   - Any directories specified using the –I option (in the order specified)
   - The directory containing the source file
   - The current directory

   For example, the compiler applies rule (1) to the following statements:

```
INCLUDE '/bob/include/file1' (absolute path name)
INCLUDE '../../file1' (relative path name)
```

   and rule (2) to this statement:

```
INCLUDE 'file1'
```

Related options: –Mnostdinc

–i2, –i4 and –i8

Treat INTEGER and LOGICAL variables as either two, four, or eight bytes.

Default: The compiler treats INTERGER and LOGICAL variables as four bytes.

Usage: In the following example, using the –i8 switch causes the integer variables to be treated as 64 bits.

```
$ pgf95 -i8 int.f
```

`int.f` is a function similar to this:

```
int.f
 print *, "Integer size:", bit_size(i)
 end
```

Description: Use this option to treat INTEGER and LOGICAL variables as either two, four, or eight bytes. INTEGER*8 values not only occupy 8 bytes of storage, but operations use 64 bits, instead of 32 bits.

Related options: None

–K<flag>

Requests that the compiler provide special compilation semantics.

Default: The compiler does not provide special compilation semantics.

Syntax:

–K<flag>

Where flag is one of the following:

ieee	Perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled, and on some systems a more accurate math library is linked if –Kieee is used during the link step.
noieee	Default flag. Use the fastest available means to perform floating-point operations, link in faster non-IEEE libraries if available, and disable underflow traps.
PIC	(Linux only) Generate position-independent code. Equivalent to –fpic. Provided for compatibility with other compilers.
pic	(Linux only) Generate position-independent code. Equivalent to –fpic. Provided for compatibility with other compilers.
trap=option [,option]...	Controls the behavior of the processor when floating-point exceptions occur. Possible options include: • fp • align (ignored) • inv • denorm • divz • ovf • unf • inexact

Usage: In the following example, the compiler performs floating-point operations in strict conformance with the IEEE 754 standard

```
$ pgf95 -Kieee myprog.f
```

Description: Use `-K` to instruct the compile to provide special compilation semantics.

The default is `-Knoieee`.

`-Ktrap` is only processed by the compilers when compiling main functions or programs. The options inv, denorm, divz, ovf, unf, and inexact correspond to the processor's exception mask bits: invalid operation, denormalized operand, divide-by-zero, overflow, underflow, and precision, respectively. Normally, the processor's exception mask bits are *on*, meaning that floating-point exceptions are masked—the processor recovers from the exceptions and continues. If a floating-point exception occurs and its corresponding mask bit is *off*, or "unmasked", execution terminates with an arithmetic exception (C's SIGFPE signal).

`-Ktrap=fp` is equivalent to `-Ktrap=inv,divz,ovf`.

Note

The PGI compilers do not support exception-free execution for`-Ktrap=inexact`. The purpose of this hardware support is for those who have specific uses for its execution, along with the appropriate signal handlers for handling exceptions it produces. It is not designed for normal floating point operation code support.

Related options: None.

--keeplnk

(Windows only.) Preserves the temporary file when the compiler generates a temporary indirect file for a long linker command.

Usage: In the following example the compiler preserves each temporary file rather than deleting it.

```
$ pgf95 --keeplnk myprog.f
```

Description: If the compiler generates a temporary indirect file for a long linker command, use this option to instruct the compiler to preserve the temporary file instead of deleting it.

Related options: None.

–L

Specifies a directory to search for libraries.

Note

Multiple –L options are valid. However, the position of multiple –L options is important relative to –l options supplied.

Syntax:

```
-Ldirectory
```

Where `directory` is the name of the library directory.

Default: The compiler searches the standard library directory.

Usage: In the following example, the library directory is `/lib` and the linker links in the standard libraries required by PGF95 from this directory.

```
$ pgf95 -L/lib myprog.f
```

In the following example, the library directory `/lib` is searched for the library file `libx.a` and both the directories `/lib` and `/libz` are searched for `liby.a`.

```
$ pgf95 -L/lib -lx -L/libz -ly myprog.f
```

Use the –L option to specify a directory to search for libraries. Using –L allows you to add directories to the search path for library files.

Related options: -l

–l<library>

Instructs the linker to load the specified library. The linker searches <library> in addition to the standard libraries.

Note

The linker searches the libraries specified with –l in order of appearance *before* searching the standard libraries.

Syntax:

```
-llibrary
```

Where library is the name of the library to search.

Usage: In the following example, if the standard library directory is `/lib` the linker loads the library `/lib/libmylib.a`, in addition to the standard libraries.

```
$ pgf95 myprog.f -lmylib
```

Description: Use this option to instruct the linker to load the specified library. The compiler prepends the characters lib to the library name and adds the .a extension following the library name. The linker searches each library specifies before searching the standard libraries.

Related options: –L

–m

Displays a link map on the standard output.

Default: The compiler does display the link map and does not use the –m option.

Usage: When the following example is executed on Windows, pgf95 creates a link map in the file `myprog.map`.

```
$ pgf95 -m myprog.f
```

Description: Use this option to display a link map.

- On Linux, the map is written to `stdout`.
- On Windows, the map is written to a `.map` file whose name depends on the executable. If the executable is `myprog.f`, the map file is in `myprog.map`.

Related options: –c, –o, -s, –u

–M<pgflag>

Selects options for code generation. The options are divided into the following categories:

Code generation	Fortran Language Controls	Optimization
Environment	C/C++ Language Controls	Miscellaneous
Inlining		

The following table lists and briefly describes the options alphabetically and includes a field showing the category. For more details about the options as they relate to these categories, refer to "–M Options by Category," on page 253.

Table 16.7. –M Options Summary

pgflag	Description	Category
allocatable=95|03	Controls whether to use Fortran 95 or Fortran 2003 semantics in allocatable array assignments.	Fortran Language
anno	Annotate the assembly code with source code.	Miscellaneous
[no]autoinline	C/C++ when a function is declared with the inline keyword, inline it at –O2 and .	Inlining
[no]asmkeyword	Specifies whether the compiler allows the asm keyword in C/C++ source files (pgcc and pgcpp only).	C/C++ Language
[no]backslash	Determines how the backslash character is treated in quoted strings (pgf77, pgf95, and pghpf only).	Fortran Language
[no]bounds	Specifies whether array bounds checking is enabled or disabled.	Miscellaneous
--[no_]builtin	Do/don't compile with math subroutine builtin support, which causes selected math library routines to be inlined (pgcc and pgcpp only).	Optimization
byteswapio	Swap byte-order (big-endian to little-endian or vice versa) during I/O of Fortran unformatted data.	Miscellaneous
cache_align	Where possible, align data objects of size greater than or equal to 16 bytes on cache-line boundaries.	Optimization
chkfpstk	Check for internal consistency of the x87 FP stack in the prologue of a function and after returning from a function or subroutine call (–tp px/p5/p6/piii targets only).	Miscellaneous
chkptr	Check for NULL pointers (pgf95 and pghpf only).	Miscellaneous
chkstk	Check the stack for available space upon entry to and before the start of a parallel region. Useful when many private variables are declared.	Miscellaneous
concur	Enable auto-concurrentization of loops. Multiple processors or cores will be used to execute parallelizable loops.	Optimization
cpp	Run the PGI cpp-like preprocessor without performing subsequent compilation steps.	Miscellaneous
cray	Force Cray Fortran (CF77) compatibility (pgf77, pgf95, and pghpf only).	Optimization

pgflag	Description	Category
[no]daz	Do/don't treat denormalized numbers as zero.	Code Generation
[no]dclchk	Determines whether all program variables must be declared (pgf77, pgf95, and pghpf only).	Fortran Language
[no]defaultunit	Determines how the asterisk character ("*") is treated in relation to standard input and standard output (regardless of the status of I/O units 5 and 6, pgf77, pgf95, and pghpf only).	Fortran Language
[no]depchk	Checks for potential data dependencies.	Optimization
[no]dse	Enables [disables] dead store elimination phase for programs making extensive use of function inlining.	Optimization
[no]dlines	Determines whether the compiler treats lines containing the letter "D" in column one as executable statements (pgf77, pgf95, and pghpf only).	Fortran Language
dll	Link with the DLL version of the runtime libraries (Windows only).	Miscellaneous
dollar,char	Specifies the character to which the compiler maps the dollar sign code (pgf77, pgf95, and pghpf only).	Fortran Language
[no]dwarf	Specifies not to add DWARF debug information.	Code Generation
dwarf1	When used with –g, generate DWARF1 format debug information.	Code Generation
dwarf2	When used with –g, generate DWARF2 format debug information.	Code Generation
dwarf3	When used with –g, generate DWARF3 format debug information.	Code Generation
extend	Instructs the compiler to accept 132-column source code; otherwise it accepts 72-column code (pgf77, pgf95, and pghpf only).	Fortran Language
extract	invokes the function extractor.	Inlining
[no]m128	Instructs the compiler to treat floating-point constants as float data types (pgcc and pgcpp only).	C/C++ Language
fixed	Instructs the compiler to assume F77-style fixed format source code (pgf95 and pghpf only).	Fortran Language
[no]flushz	Do/don't set SSE flush-to-zero mode	Code Generation
[no]fpapprox	Specifies not to use low-precision fp approximation operations.	Optimization
[no]f[=option]	Perform certain floating point intrinsic functions using relaxed precision.	Optimization

pgflag	Description	Category
free	Instructs the compiler to assume F90-style free format source code (pgf95 and pghpf only).	Fortran Language
func32	The compiler aligns all functions to 32-byte boundaries.	Code Generation
gccbug[s]	Matches behavior of certain gcc bugs	Miscellaneous
info	Prints informational messages regarding optimization and code generation to standard output as compilation proceeds.	Miscellaneous
inform	Specifies the minimum level of error severity that the compiler displays.	Miscellaneous
inline	Invokes the function inliner.	Inlining
instrumentation	Generates code to enable instrumentation of functions.	Miscellaneous
[no]ipa	Invokes interprocedural analysis and optimization.	Optimization
[no]iomutex	Determines whether critical sections are generated around Fortran I/O calls (pgf77, pgf95, and pghpf only).	Fortran Language
keepasm	Instructs the compiler to keep the assembly file.	Miscellaneous
largeaddressaware	Enables support for 64-bit indexing and single static data objects of size larger than 2GB.	Code Generation
[no]large_arrays	Enables support for 64-bit indexing and single static data objects of size larger than 2GB.	Code Generation
lfs	Links in libraries that allow file I/O to files of size larger than 2GB on 32-bit systems (32-bit Linux only).	Environment
[no]loop32	Aligns/does not align innermost loops on 32 byte boundaries with –tp barcelona	Code Generation
[no]lre	Disable/enable loop-carried redundancy elimination.	Optimization
list	Specifies whether the compiler creates a listing file.	Miscellaneous
[no]m128	Recognizes [ignores] __m128, __m128d, and __m128i datatypes. (C only)	Code Generation
makedll	Generate a dynamic link library (DLL) (Windows only).	Miscellaneous
makeimplib	Passes the -def switch to the librarian without a deffile, when used without `-def:deffile`.	Miscellaneous
mpi=option	Link to MPI libraries: MPICH1, MPICH2, or Microsoft MPI libraries	Code Generation

pgflag	Description	Category
neginfo	Instructs the compiler to produce information on why certain optimizations are not performed.	Miscellaneous
noframe	Eliminates operations that set up a true stack frame pointer for functions.	Optimization
noi4	Determines how the compiler treats INTEGER variables (pgf77, pgf95, and pghpf only).	Optimization
nomain	When the link step is called, don't include the object file that calls the Fortran main program. (pgf77, pgf95, and pghpf only).	Code Generation
noopenmp	When used in combination with the –mp option, the compiler ignores OpenMP parallelization directives or pragmas, but still processes SGI-style parallelization directives or pragmas.	Miscellaneous
nopgdllmain	Do not link the module containing the default DllMain() into the DLL (Windows only).	Miscellaneous
norpath	On Linux, do not add –rpath paths to the link line.	Miscellaneous
nosgimp	When used in combination with the –mp option, the compiler ignores SGI-style parallelization directives or pragmas, but still processes OpenMP directives or pragmas.	Miscellaneous
[no]stddef	Instructs the compiler to not recognize the standard preprocessor macros.	Environment
nostdinc	Instructs the compiler to not search the standard location for include files.	Environment
nostdlib	Instructs the linker to not link in the standard libraries.	Environment
[no]onetrip	Determines whether each DO loop executes at least once (pgf77, pgf95, and pghpf only).	Language
novintr	Disable idiom recognition and generation of calls to optimized vector functions.	Optimization
pfi	Instrument the generated code and link in libraries for dynamic collection of profile and data information at runtime.	Optimization
pre	Read a pgfi.out trace file and use the information to enable or guide optimizations.	Optimization
pre	Force/disable generation of non-temporal moves and prefetching.	Code Generation
[no]prefetch	Enable/disable generation of prefetch instructions.	Optimization

pgflag	Description	Category
preprocess	Perform cpp-like preprocessing on assembly language and Fortran input source files.	Miscellaneous
prof	Set profile options; function-level and line-level profiling are supported.	Code Generation
[no]r8	Determines whether the compiler promotes REAL variables and constants to DOUBLE PRECISION (pgf77, pgf95, and pghpf only).	Optimization
[no]r8intrinsics	Determines how the compiler treats the intrinsics CMPLX and REAL (pgf77, pgf95, and pghpf only).	Optimization
[no]recursive	Allocate / do not allocate local variables on the stack, this allows recursion. SAVEd, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch (pgf77, pgf95, and pghpf only).	Code Generation
[no]reentrant	Specifies whether the compiler avoids optimizations that can prevent code from being reentrant.	Code Generation
[no]ref_externals	Do/don't force references to names appearing in EXTERNAL statements (pgf77, pgf95, and pghpf only).	Code Generation
safeptr	Instructs the compiler to override data dependencies between pointers and arrays (pgcc and pgcpp only).	Optimization
safe_lastval	In the case where a scalar is used after a loop, but is not defined on every iteration of the loop, the compiler does not by default parallelize the loop. However, this option tells the compiler it is safe to parallelize the loop. For a given loop, the last value computed for all scalars make it safe to parallelize the loop.	Code Generation
[no]save	Determines whether the compiler assumes that all local variables are subject to the SAVE statement (pgf77, pgf95, and pghpf only).	Fortran Language
[no]scalarsse	Do/don't use SSE/SSE2 instructions to perform scalar floating-point arithmetic.	Optimization
schar	Specifies signed char for characters (pgcc and pgcpp only - also see uchar).	C/C++ Language
[no]second_underscore	Do/don't add the second underscore to the name of a Fortran global if its name already contains an underscore (pgf77, pgf95, and pghpf only).	Code Generation
[no]signextend	Do/don't extend the sign bit, if it is set.	Code Generation

pgflag	Description	Category
[no]single	Do/don't convert float parameters to double parameter characters (pgcc and pgcpp only).	C/C++ Language
[no]smart	Do/don't enable optional post-pass assembly optimizer.	Optimization
[no]smartalloc[=huge|huge:<n>|hugebss]	Add a call to the routine mallopt in the main routine. Supports large TLBs on Linux and Windows. *Tip.* To be effective, this switch must be specified when compiling the file containing the Fortran, C, or C++ main program.	Environment
standard	Causes the compiler to flag source code that does not conform to the ANSI standard(pgf77, pgf95, and pghpf only).	Fortran Language
[no]stride0	Do/do not generate alternate code for a loop that contains an induction variable whose increment may be zero (pgf77, pgf95, and pghpf only).	Code Generation
uchar	Specifies unsigned char for characters (pgcc and pgcpp only - also see schar).	C/C++ Language
unix	Uses UNIX calling and naming conventions for Fortran subprograms (pgf77, pgf95, and pghpf for Win32 only).	Code Generation
[no]unixlogical	Determines how the compiler treats logical values. (pgf77, pgf95, and pghpf only).	Fortran Language
[no]unroll	Controls loop unrolling.	Optimization
[no]upcase	Determines whether the compiler preserves uppercase letters in identifiers. (pgf77, pgf95, and pghpf only).	Fortran Language
varargs	Forces Fortran program units to assume calls are to C functions with a varargs type interface (pgf77 and pgf95 only).	Code Generation
[no]vect	Do/don't invoke the code vectorizer.	Optimization

–mcmodel=medium

(For use only on 64-bit Linux targets) Generates code for the medium memory model in the linux86-64 execution environment. Implies –Mlarge_arrays.

Default: The compiler generates code for the small memory model.

Usage: The following command line requests position independent code be generated, and the option –`mcmodel=medium` be passed to the assembler and linker:

```
$ pgf95 -mcmodel=medium myprog.f
```

Description: The default small memory model of the linux86-64 environment limits the combined area for a user's object or executable to 1GB, with the Linux kernel managing usage of the second 1GB of address for system routines, shared libraries, stacks, and so on. Programs are started at a fixed address, and the program can use a single instruction to make most memory references.

The medium memory model allows for larger than 2GB data areas, or .bss sections. Program units compiled using either `–mcmodel=medium` or `–fpic` require additional instructions to reference memory. The effect on performance is a function of the data-use of the application. The `–mcmodel=medium` switch must be used at both compile time and link time to create 64-bit executables. Program units compiled for the default small memory model can be linked into medium memory model executables as long as they are compiled with the option`–fpic`, or position-independent.

The linux86-64 environment provides `static libxxx.a` archive libraries, that are built both with and without `–fpic`, and `dynamic libxxx.so` shared object libraries that are compiled `–fpic`. Using the link switch `–mcmodel=medium` implies the `–fpic` switch and utilizes the shared libraries by default. The `$PGI/linux86-64/<rel>/lib` directory contains the libraries for building small memory model codes; and the `$PGI/linux86-64/<rel>/libso` directory contains shared libraries for building both `–fpic` and `Žmcmodel=medium` executables.

Note

`–mcmodel=medium -fpic` is not allowed to create shared libraries. However, you can create static archive libraries (.a) that are `–fpic`.

Related options:–Mlarge_arrays

–module <moduledir>

Allows you to specify a particular directory in which generated intermediate `.mod` files should be placed.

Default: The compiler places `.mod` files in the current working directory, and searches only in the current working directory for pre-compiled intermediate `.mod` files.

Usage: The following command line requests that any intermediate module file produced during compilation of `myprog.f` be placed in the directory `mymods`; specifically, the file `./mymods/myprog.mod` is used.

```
$ pgf95 -module mymods myprog.f
```

Description: Use the –module option to specify a particular directory in which generated intermediate .mod files should be placed. If the –module <moduledir> option is present, and USE statements are present in a compiled program unit, then <moduledir> is searched for `.mod` intermediate files *prior* to a search in the default local directory.

Related options: None.

–mp[=all, align,bind,[no]numa]

Instructs the compiler to interpret user-inserted OpenMP shared-memory parallel programming directives and pragmas, and to generate an executable file which will utilize multiple processors in a shared-memory parallel system.

Default: The compiler ignores user-inserted shared-memory parallel programming directives and pragmas.

Usage: The following command line requests processing of any shared-memory directives present in `myprog.f`:

```
$ pgf95 -mp myprog.f
```

Description: Use the `-mp`option to instruct the compiler to interpret user-inserted OpenMP shared-memory parallel programming directives and to generate an executable file which utilizes multiple processors in a shared-memory parallel system.

The sub-options are one or more of the following:

align
: Forces loop iterations to be allocated to OpenMP processes using an algorithm that maximizes alignment of vector sub-sections in loops that are both parallelized and vectorized for SSE. This allocation can improve performance in program units that include many such loops. It can also result in load-balancing problems that significantly decrease performance in program units with relatively short loops that contain a large amount of work in each iteration. The numa suboption uses libnuma on systems where it is available.

allcores
: Instructs the compiler to all available cores. You specify this sub-option at link time.

bind
: Instructs the compiler to bind threads to cores. You specify this sub-option at link time.

[no]numa
: Uses [does not use] libnuma on systems where it is available.

For a detailed description of this programming model and the associated directives and pragmas, refer to Chapter 5, "*Using OpenMP*".

Related options: –Mconcur, –Mvect

–noswitcherror

Issues warnings instead of errors for unknown switches. Ignores unknown command line switches after printing a warning message.

Default: The compiler prints an error message and then halts.

Usage: In the following example, the compiler ignores unknown command line switches after printing a warning message.

```
$ pgf95 -noswitcherror myprog.f
```

Description: Use this option to instruct the compiler to ignore unknown command line switches after printing an warning message.

Tip

You can configure this behavior in the `siterc` file by adding: `set NOSWITCHERROR=1.`

Related options: None.

–O<level>

Invokes code optimization at the specified level.

Default: The compiler optimizes at level 2.

Syntax:

–O [level]

Where level is an integer from 0 to 4.

Usage: In the following example, since no –O option is specified, the compiler sets the optimization to level 1.

```
$ pgf95 myprog.f
```

In the following example, since no optimization level is specified and a –O option is specified, the compiler sets the optimization to level 2.

```
$ pgf95 -O myprog.f
```

Description: Use this option to invoke code optimization at the specified level - one of the following:

0
: creates a basic block for each statement. Neither scheduling nor global optimization is done. To specify this level, supply a 0 (zero) argument to the –O option.

1
: schedules within basic blocks and performs some register allocations, but does no global optimization.

2
: performs all level-1 optimizations, and also performs global scalar optimizations such as induction variable elimination and loop invariant movement.

3
: level-three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

4
: level-four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

Table 16.8 shows the interaction between the `-O` option, `-g` option, `-Mvect`, and `-Mconcur` options.

Table 16.8. Optimization and –O, –g, –Mvect, and –Mconcur Options

Optimize Option	Debug Option	–M Option	Optimization Level
none	none	none	1
none	none	–Mvect	2

Optimize Option	Debug Option	–M Option	Optimization Level
none	none	–Mconcur	2
none	–g	none	0
–O	none or –g	none	2
–Olevel	none or –g	none	level
–Olevel < 2	none or –g	–Mvect	2
–Olevel < 2	none or –g	–Mconcur	2

Unoptimized code compiled using the option `-O0` can be significantly slower than code generated at other optimization levels. Like the `-Mvect` option, the `-Munroll` option sets the optimization level to level-2 if no –O or `-g` options are supplied. The `-gopt` option is recommended for generation of debug information with optimized code. For more information on optimization, see Chapter 3, "*Optimizing & Parallelizing*".

Related options: –g, –M<pgflag>, –gopt

–o

Names the executable file. Use the –o option to specify the filename of the compiler object file. The final output is the result of linking.

Syntax:

–o filename

Where filename is the name of the file for the compilation output. The filename must not have a .f extension.

Default: The compiler creates executable filenames as needed. If you do not specify the `-o` option, the default filename is the linker output file `a.out`.

Usage: In the following example, the executable file is `myprog` instead of the default `a.outmyprog.exe`.

```
$ pgf95 myprog.f -o myprog
```

Related options: –c, –E, –F, –S

–pc

Note

This option is available only for –tp px/p5/p6/piii targets.

Allows you to control the precision of operations performed using the x87 floating point unit, and their representation on the x87 floating point stack.

Syntax:

–pc { 32 | 64 | 80 }

Usage:

```
$ pgf95 -pc 64 myprog.f
```

Description: The x87 architecture implements a floating-point stack using 8 80-bit registers. Each register uses bits 0-63 as the significant, bits 64-78 for the exponent, and bit 79 is the sign bit. This 80-bit real format is the default format, called the *extended format*. When values are loaded into the floating point stack they are automatically converted into extended real format. The precision of the floating point stack can be controlled, however, by setting the precision control bits (bits 8 and 9) of the floating control word appropriately. In this way, you can explicitly set the precision to standard IEEE double-precision using 64 bits, or to single precision using 32 bits.[1] The default precision is system dependent. To alter the precision in a given program unit, the main program must be compiled with the same -pc option. The command line option –pc val lets the programmer set the compiler's precision preference.

Valid values for val are:

32 single precision 64 double precision 80 extended precision

Extended Precision Option – Operations performed exclusively on the floating-point stack using extended precision, without storing into or loading from memory, can cause problems with accumulated values within the extra 16 bits of extended precision values. This can lead to answers, when rounded, that do not match expected results.

For example, if the argument to `sin` is the result of previous calculations performed on the floating-point stack, then an 80-bit value used instead of a 64-bit value can result in slight discrepancies. Results can even change sign due to the sin curve being too close to an x-intercept value when evaluated. To maintain consistency in this case, you can assure that the compiler generates code that calls a function. According to the x86 ABI, a function call must push its arguments on the stack (in this way memory is guaranteed to be accessed, even if the argument is an actual constant). Thus, even if the called function simply performs the inline expansion, using the function call as a wrapper to sin has the effect of trimming the argument precision down to the expected size. Using the `-Mnobuiltin` option on the command line for C accomplishes this task by resolving all math routines in the library `libm`, performing a function call of necessity. The other method of generating a function call for math routines, but one that may still produce the inline instructions, is by using the `-Kieee` switch.

A second example illustrates the precision control problem using a section of code to determine machine precision:

```
 program find_precision

 w = 1.0
100 w=w+w
 y=w+1
 z=y-w
 if (z .gt. 0) goto 100
C now w is just big enough that |((w+1)-w)-1| >= 1
...
 print*,w
 end
```

In this case, where the variables are implicitly real*4, operations are performed on the floating-point stack where optimization removes unnecessary loads and stores from memory. The general case of copy propagation being performed follows this pattern:

[1]According to Intel documentation, this only affects the x87 operations of add, subtract, multiply, divide, and square root. In particular, it does not appear to affect the x87 transcendental instructions.

```
a = x
y = 2.0 + a
```

Instead of storing x into a, then loading a to perform the addition, the value of x can be left on the floating-point stack and added to 2.0. Thus, memory accesses in some cases can be avoided, leaving answers in the extended real format. If copy propagation is disabled, stores of all left-hand sides will be performed automatically and reloaded when needed. This will have the effect of rounding any results to their declared sizes.

When executed using default (extended) precision, the find_precision program has a value of 1.8446744E +19. If, however, `-Kieee` is set, the value becomes 1.6777216E+07 (single precision.) This difference is due to the fact that `-Kieee` disables copy propagation, so all intermediate results are stored into memory, then reloaded when needed. Copy propagation is only disabled for floating-point operations, not integer. With this particular example, setting the `-pc` switch will also adjust the result.

The `-Kieee` switch also has the effect of making function calls to perform all transcendental operations. Except when the `-Mnobuiltin` switch is set in C, the function still produces the x86 machine instruction for computation, and arguments are passed on the stack, which results in a memory store and load.

Finally, `-Kieee` also disables reciprocal division for constant divisors. That is, for a/b with unknown a and constant b, the expression is usually converted at compile time to a*(1/b), thus turning an expensive divide into a relatively fast scalar multiplication. However, numerical discrepancies can occur when this optimization is used.

Understanding and correctly using the `-pc`, `-Mnobuiltin`, and `-Kieee` switches should enable you to produce the desired and expected precision for calculations which utilize floating-point operations.

Related options: –Kieee, –Mnobuiltin

- -pedantic

Prints warnings from included <system header files> .

Syntax:

-- pedantic

Default: The compiler prints the warnings from the included system header files.

Usage: In the following example, the compiler prints the warnings from the included system header files.

```
$ pgf95 --pedantic myprog.f
```

Related options:

–pg

(Linux only) Instructs the compiler to instrument the generated executable for gprof-style sample-based profiling.

Usage: In the following example the program is compiled for profiling using pgdbg or gprof.

```
$ pgf95 -pg myprog.c
```

Default: The compiler does not instrument the generated executable for gprof-style profiling.

Description: Use this option to instruct the compiler to instrument the generated executable for gprof-style sample-based profiling. You must use this option at both the compile and link steps. A `gmon.out` style trace is generated when the resulting program is executed, and can be analyzed using gprof or pgprof.

–pgcpplibs

Instructs the compiler to append C++ runtime libraries to the link line for programs built using either PGF90 or PGF77.

Default: The C/C++ compilers do not append the C++ runtime libraries to the link line.

Usage: In the following example the C++ runtime libraries are linked with an object file compiled with pgf77.

```
$ pgf90 main.f90 mycpp.o -pgcpplibs
```

Description: Use this option to instruct the compiler to append C++ runtime libraries to the link line for programs built using either PGF90 or PGF77.

Related options:–pgf90libs, –pgf77libs

–pgf77libs

Instructs the compiler to append PGF77 runtime libraries to the link line.

Default: The C/C++ compilers do not append the PGF77 runtime libraries to the link line.

Usage: In the following example a .c main program is linked with an object file compiled with pgf77.

```
$ pgcc main.c myf77.o -pgf77libs
```

Description: Use this option to instruct the compiler to append PGF77 runtime libraries to the link line.

Related options:–pgcpplibs, –pgf90libs

–pgf90libs

Instructs the compiler to append PGF90/PGF95 runtime libraries to the link line.

Default: The C/C++ compilers do not append the PGF90/PGF95 runtime libraries to the link line.

Usage: In the following example a .c main program is linked with an object file compiled with pgf95.

```
$ pgcc main.c myf95.o -pgf90libs
```

Description: Use this option to instruct the compiler to append PGF90/PGF95 runtime libraries to the link line.

Related options:–pgcpplibs, –pgf77libs

–R<directory>

(Linux only) Instructs the linker to hard-code the pathname <directory> into the search path for generated shared object (dynamically linked library) files.

Note

There cannot be a space between R and <directory>.

Usage: In the following example, at runtime the a.out executable searches the specified directory, in this case `/home./Joe/myso`, for shared objects.

```
$ pgf95 -Rm/home/Joe/myso myprog.f
```

Description: Use this option to instruct the compiler to pass information to the linker to hard-code the pathname <directory> into the search path for shared object (dynamically linked library) files.

Related options: –fpic, –shared, –G

–r

Linux only. Creates a relocatable object file.

Default: The compiler does not create a relocatable object file and does not use the –r option.

Usage: In this example, pgf95 creates a relocatable object file.

```
$ pgf95 -r myprog.f
```

Description: Use this option to create a relocatable object file.

Related options: –c, –o, –s, –u

–r4 and –r8

Interprets DOUBLE PRECISION variables as REAL (–r4), or interprets REAL variables as DOUBLE PRECISION (–r8).

Usage: In this example, the double precision variables are interpreted as REAL.

```
$ pgf95 -r4 myprog.f
```

Description: Interpret DOUBLE PRECISION variables as REAL (–r4) or REAL variables as DOUBLE PRECISION (–r8).

Related options: –i2, –i4, –i8, –nor8

–rc

Specifies the name of the driver startup configuration file. If the file or pathname supplied is not a full pathname, the path for the configuration file loaded is relative to the $DRIVER path (the path of the currently executing driver). If a full pathname is supplied, that file is used for the driver configuration file.

Syntax:

```
-rc [path] filename
```

Where path is either a relative pathname, relative to the value of $DRIVER, or a full pathname beginning with "/". Filename is the driver configuration file.

Default: The driver uses the configuration file `.pgirc`.

Usage: In the following example, the file `.pgf95rctest`, relative to `/usr/pgi/linux86/bin`, the value of $DRIVER, is the driver configuration file.

```
$ pgf95 -rc .pgf95rctest myprog.f
```

Description: Use this option to specify the name of the driver startup configuration file. If the file or pathname supplied is not a full pathname, the path for the configuration file loaded is relative to the $DRIVER path - the path of the currently executing driver. If a full pathname is supplied, that file is used for the driver configuration file.

Related options: –show

–rpath

(Linux only) Specifies the name of the driver startup configuration file.

Syntax:

```
-rpath path <ldarg>
```

where path is either a relative pathname, or a full pathname beginning with "/".

Default: The driver uses the configuration file `.pgirc`.

Usage: In the following example, the file `.pgf95rctest`, relative to `/usr/pgi/linux86/bin`, the value of $DRIVER, is the driver configuration file.

```
$ pgf95 -rc .pgf95rctest myprog.f
```

Description: Use this option to specify the name of the driver startup configuration file. If the file or pathname supplied is not a full pathname, the path for the configuration file loaded is relative to the $DRIVER path - the path of the currently executing driver. If a full pathname is supplied, that file is used for the driver configuration file.

With the ldarg option (Linux only), the ldarg information is passed to the linker and the directory is added to the runtime shared library search path.

Related options: –show

–s

(Linux only) Strips the symbol-table information from the executable file.

Default: The compiler includes all symbol-table information and does not use the –s option.

Usage: In this example, pgf95 strips symbol-table information from the `a.out`. executable file.

```
$ pgf95 -s myprog.f
```

Description: Use this option to strip the symbol-table information from the executable.

Related options: –c, –o, –u

–S

Stops compilation after the compiling phase and writes the assembly-language output to a file.

Default: The compiler does not produce a `.s` file.

Usage: In this example, pgf95 produces the file `myprog.s` in the current directory.

```
$ pgf95 -S myprog.f
```

Description: Use this option to stop compilation after the compiling phase and then write the assembly-language output to a file. If the input file is `filename.f`, then the output file is `filename.s`.

Related options: –c, –E, –F, –Mkeepasm, –o

–shared

(Linux only) Instructs the compiler to pass information to the linker to produce a shared object (dynamically linked library) file.

Default: The compiler does not pass information to the linker to produce a shared object file.

Usage: In the following example the compiler passes information to the linker to produce the shared object file: `myso.so`.

```
$ pgf95 -shared myprog.f -o myso.so
```

Description: Use this option to instruct the compiler to pass information to the linker to produce a shared object (dynamically linked library) file.

Related options: –fpic, –G, –R

–show

Produces driver help information describing the current driver configuration.

Default: The compiler does not show driver help information.

Usage: In the following example, the driver displays configuration information to the standard output after processing the driver configuration file.

```
$ pgf95 -show myprog.f
```

Description: Use this option to produce driver help information describing the current driver configuration.

Related options: –V, –v, –###, –help, –rc

–silent

Do not print warning messages.

Default: The compiler prints warning messages.

Usage: In the following example, the driver does not display warning messages.

```
$ pgf95 -silent myprog.f
```

Description: Use this option to suppress warning messages.

Related options: –v, –V, –w

–soname

(Linux only.) The compiler recognizes the –soname option and passes it through to the linker.

Default: The compiler does not recognize the –soname option.

Usage: In the following example, the driver passes the soname option and its argument through to the linker.

```
$ pgf95 -soname library.so myprog.f
```

Description: Use this option to instruct the compiler to recognize the –soname option and pass it through to the linker.

Related options:

–stack

(Windows only.) Allows you to explicitly set stack properties for your program.

Default: If `-stack` is not specified, then the defaults are as followed:

Win32
: Setting is `-stack:2097152,2097152`, which is approximately 2MB for reserved and committed bytes.

Win64
: No default setting

Syntax:

```
-stack={ (reserved bytes)[,(committed bytes)] }{, [no]check }
```

Usage: The following example demonstrates how to reserve 524,288 stack bytes (512KB), commit 262,144 stack bytes for each routine (256KB), and disable the stack initialization code with the nocheck argument.

```
$ pgf95 -stack=524288,262144,nocheck myprog.f
```

Description: Use this option to explicitly set stack properties for your program. The `-stack` option takes one or more arguments: (reserved bytes), (committed bytes), [no]check.

reserved bytes
: Specifies the total stack bytes required in your program.

committed bytes
: Specifies the number of stack bytes that the Operating System will allocate for each routine in your program. This value must be less than or equal to the stack *reserved bytes* value.

  Default for this argument is 4096 bytes

[no]check
: Instructs the compiler to generate or not to generate stack initialization code upon entry of each routine. Check is the default, so stack initialization code is generated.

Stack initialization code is required when a routine's stack exceeds the *committed bytes* size. When your *committed bytes* is equal to the *reserved bytes* or equal to the stack bytes required for each routine, then

you can turn off the stack initialization code using the `-stack=nocheck` compiler option. If you do this, the compiler assumes that you are specifying enough committed stack space; and therefore, your program does not have to manage its own stack size.

For more information on determining the amount of stack required by your program, refer to `–Mchkstk` compiler option, described in "Miscellaneous Controls".

Note

`-stack=(reserved bytes),(committed bytes)` are linker options.

`-stack=[no]check` is a compiler option.

If you specify `-stack=(reserved bytes),(committed bytes)` on your compile line, it is only used during the link step of your build. Similarly, `-stack=[no]check` can be specified on your link line, but its only used during the compile step of your build.

Related options: –Mchkstk

–ta=nvidia(,nvidia_suboptions),host

Defines the target accelertator.

Note

this flag is valid only for Fortran and C.

Default: The compiler uses NVIDIA .

Usage: In the following example, NVIDEA is the accelerator target architecture and the accelerator generates code for compute capability 1.3.

```
$ pgf95 -ta=nvidia(cc13)
```

Description: Use this option to select the accelerator target and, optionally, to define the type of code to genertate.

The `–ta` flag has the following options:

nvidia
: Select NVIDIA accelerator target. This option has the following nvidia-suboptions:

  analysis
  : Perform loop analysis only; do not generate GPU code.

  cc10
  : Generate code for compute capability 1.0.

  cc11
  : Generate code for compute capability 1.1.

  cc13
  : Generate code for compute capability 1.3.

nofma
: Do not generate fused multiply-add instructions.

time
: Link in a limited-profiling library, as described in "Profiling Accelerator Kernels," on page 96.

host
: Select NO accelerator target. Generate PGI Unified Binary Code, as described "PGI Unified Binary for Accelerators," on page 95.

Related options: –#

–time

Print execution times for various compilation steps.

Default: The compiler does not print execution times for compilation steps.

Usage: In the following example, pgf95 prints the execution times for the various compilation steps.

```
$ pgf95 -time myprog.f
```

Description: Use this option to print execution times for various compilation steps.

Related options: –#

–tp <target> [,target...]

Sets the target architecture.

Default: The PGI compilers produce code specifically targeted to the type of processor on which the compilation is performed. In particular, the default is to use all supported instructions wherever possible when compiling on a given system.

The default style of code generation is auto-selected depending on the type of processor on which compilation is performed. Further, the –tp x64 style of unified binary code generation is only enabled by an explicit –tp x64 option.

Note

Executables created on a given system may not be usable on previous generation systems. (For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.)

Usage: In the following example, pgf95 sets the target architecture to EM64T:

```
$ pgf95 -tp p7-64 myprog.f
```

Description: Use this option to set the target architecture. By default, the PGI compiler uses all supported instructions wherever possible when compiling on a given system. As a result, executables created on a given system may not be usable on previous generation systems. For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.

Processor-specific optimizations can be specified or limited explicitly by using the –tp option. Thus, it is possible to create executables that are usable on previous generation systems. With the exception of k8-64,

k8-64e, p7-64, and x64, any of these sub-options are valid on any x86 or x64 processor-based system. The k8-64, k8-64e, p7-64 and x64 options are valid only on x64 processor-based systems.

The –tp x64 option generates unified binary object and executable files, as described in the section called "Using –tp to Generate a Unified Binary".

The following list contains the possible sub-options for –tp and the processors that each sub-option is intended to target:

athlon
: generate 32-bit code for AMD Athlon XP/MP and compatible processors.

barcelona
: generate 32-bit code for AMD Opteron/Quadcore and compatible processors.

barcelona-32
: generate 32-bit code for AMD Opteron/Quadcore and compatible processors. Same as bareclona suboption.

barcelona-64
: generate 64-bit code for AMD Opteron/Quadcore and compatible processors.

core2
: generate 32-bit code for Intel Core 2 Duo and compatible processors.

core2-32
: generate 32-bit code for Intel Core 2 Duo and compatible processors. Same as core2 option.

core2-64
: generate 64-bit code for Intel Core 2 Duo EM64T and compatible processors.

istanbul
: generate 32-bit code that is usable on any Istanbul processor-based system.

istanbul-32
: generate 32-bit code that is usable on any Istanbul processor-based system.

istanbul-64
: generate 64-bit code that is usable on any Istanbul processor-based system.

k8-32
: generate 32-bit code for AMD Athlon64, AMD Opteron and compatible processors.

k8-64
: generate 64-bit code for AMD Athlon64, AMD Opteron and compatible processors.

k8-64e
: generate 64-bit code for AMD Opteron Revision E, AMD Turion, and compatible processors.

nehalem
: generate 32-bit code that is usable on any Nehalem processor-based system.

nehalem-32
: generate 32-bit code that is usable on any Nehalem processor-based system.

nehalem-64
: generate 64-bit code that is usable on any Nehalem processor-based system.

p6
: generate 32-bit code for Pentium Pro/II/III and AthlonXP compatible processors.

p7
: generate 32-bit code for Pentium 4 and compatible processors.

p7-32
: generate 32-bit code for Pentium 4 and compatible processors. Same as p7 option.

p7-64
: generate 64-bit code for Intel P4/Xeon EM64T and compatible processors.

penryn
: generate 32-bit code for Intel Penryn Architecture and compatible processors.

penryn-32
: generate 32-bit code for Intel Penryn Architecture and compatible processors. Same as penryn suboption.

penryn-64
: generate 64-bit code for Intel Penryn Architecture and compatible processors.

piii
: generate 32-bit code for Pentium III and compatible processors, including support for single-precision vector code using SSE instructions.

px
: generate 32-bit code that is usable on any x86 processor-based system.

px-32
: generate 32-bit code that is usable on any x86 processor-based system. Same as px suboption.

shanghai
: generate 32-bit code that is usable on any AMD Shanghai processor-based system.

shanghai-32
: generate 32-bit code that is usable on any AMD Shanghai processor-based system.

shanghai-64
: generate 64-bit code that is usable on any AMD Shanghai processor-based system.

x64
: generate 64-bit unified binary code including full optimizations and support for both AMD and Intel x64 processors.

Refer to the PGI Release Notes for a concise list of the features of these processors that distinguish them as separate targets when using the PGI compilers and tools.

The syntax for 64-bit and 32-bit targets is similar, even though the target information varies.

Syntax for 64-bit targets:

```
-tp {k8-64 | k8-64e | p7-64 | core2-64 | x64}
```

Syntax for 32-bit targets:

```
-tp {k8-32 | p6 | p7 | core2 | piii | px}
```

Using –tp to Generate a Unified Binary

Different processors have differences, some subtle, in hardware features such as instruction sets and cache size. The compilers make architecture-specific decisions about such things as instruction selection, instruction scheduling, and vectorization. Any of these decisions can have significant effects on performance and compatibility. PGI unified binaries provide a low-overhead means for a single program to run well on a number of hardware platforms.

You can use the –tp option to produce PGI Unified Binary programs. The compilers generate, and combine into one executable, multiple binary code streams, each optimized for a specific platform. At runtime, this one executable senses the environment and dynamically selects the appropriate code stream.

The target processor switch, –tp, accepts a comma-separated list of 64-bit targets and will generate code optimized for each listed target. For example, the following switch generates optimized code for three targets: k8-64, p7-64, and core2-64.

Syntax for optimizing for multiple targets:

```
-tp k8-64,p7-64,core2-64
```

The –tp k8-64 and –tp k8-64e options result in generation of code supported on and optimized for AMD x64 processors, while the –tp p7-64 option results in generation of code that is supported on and optimized for Intel x64 processors. Performance of k8-64 or k8-64e code executed on Intel x64 processors, or of p7-64 code executed on AMD x64 processors, can often be significantly less than that obtained with a native binary.

The special –tp x64 option is equivalent to –tp k8-64,p7-64. This switch produces PGI Unified Binary programs containing code streams fully optimized and supported for *both* AMD64 and Intel EM64T processors.

For more information on unified binaries, refer to "Processor-Specific Optimization & the Unified Binary," on page 39.

Related options: –M<pgflag> options that control environments

–u

Initializes the symbol-table with <symbol>, which is undefined for the linker.

Default: The compiler does not use the –u option.

Syntax:

```
-usymbol
```

Where *symbol* is a symbolic name.

Usage: In this example, pgf95 initializes symbol-table with `test`.

```
$ pgf95 -utest myprog.f
```

Description: Use this option to initialize the symbol-table with <symbol>, which is undefined for the linker. An undefined symbol triggers loading of the first member of an archive library.

Related options: –c, –o, –s

–U

Undefines a preprocessor macro.

Syntax:

```
-Usymbol
```

Where *symbol* is a symbolic name.

Usage: The following examples undefine the macro test.

```
$ pgf95 -Utest myprog.F
$ pgf95 -Dtest -Utest myprog.F
```

Description: Use this option to undefine a preprocessor macro. You can also use the #undef preprocessor directive to undefine macros.

Related options: –D, –Mnostddef.

–V[release_number]

Displays additional information, including version messages. Further, if a `release_number` is appended, the compiler driver attempts to compile using the specified release instead of the default release.

Note

There can be no space between `-V` and `release_number`.

Default: The compiler does not display version information and uses the release specified by your path to compile.

Usage: The following command-line shows the output using the `-V` option.

```
% pgf95 -V myprog.f
```

The following command-line causes PGF95 to compile using the 5.2 release instead of the default release.

```
% pgcc -V5.2 myprog.c
```

Description: Use this option to display additional information, including version messages or, if a release_number is appended, to instruct the compiler driver to attempt to compile using the specified release instead of the default release.

The specified release must be co-installed with the default release, and must have a release number greater than or equal to 4.1, which was the first release that supported this functionality.

Related options: –Minfo, –v

–v

Displays the invocations of the compiler, assembler, and linker.

Default: The compiler does not display individual phase invocations.

Usage: In the following example you use –v to see the commands sent to compiler tools, assembler, and linker.

```
$ pgf95 -v myprog.f90
```

Description: Use the –v option to display the invocations of the compiler, assembler, and linker. These invocations are command lines created by the compiler driver from the files and the –W options you specify on the compiler command-line.

Related options: –dryrun, –Minfo, –V, –W

–W

Passes arguments to a specific phase.

Syntax:

```
-W{0 | a | l },option[,option...]
```

Note

You cannot have a space between the –W and the single-letter pass identifier, between the identifier and the comma, or between the comma and the option.

0
(the number zero) specifies the compiler.

a
specifies the assembler.

l
(lowercase letter l) specifies the linker.

option
is a string that is passed to and interpreted by the compiler, assembler or linker. Options separated by commas are passed as separate command line arguments.

Usage: In the following example the linker loads the text segment at address `0xffc00000` and the data segment at address `0xffe00000`.

```
$ pgf95 -Wl,-k,-t,0xffc00000,-d,0xffe00000 myprog.f
```

Description: Use this option to pass arguments to a specific phase. You can use the –W option to specify options for the assembler, compiler, or linker.

Note

A given PGI compiler command invokes the compiler driver, which parses the command-line, and generates the appropriate commands for the compiler, assembler, and linker.

Related options: –Minfo, –V, –v

–w

Do not print warning messages.

Default: The compiler prints warning messages.

Usage: In the following example no warning messages are printed.

```
$ pgf95 -w myprog.f
```

Description: Use the –w option to not print warning messages. Sometimes the compiler issues many warning in which you may have no interest. You can use this option to not issue those warnings.

Related options: –silent

–Xs

Use legacy standard mode for C and C++.

Default: None.

Usage: In the following example the compiler uses legacy standard mode.

```
$ pgcc -Xs myprog.c
```

Description: Use this option to use legacy standard mode for C and C++. Further, this option implies -alias=traditional.

Related options: –alias, –Xt

–Xt

Use legacy transitional mode for C and C++.

Default: None.

Usage: In the following example the compiler uses legacy transitional mode.

```
$ pgcc -Xt myprog.c
```

Description: Use this option to use legacy transitional mode for C and C++. Further, this option implies -alias=traditional.

Related options: –alias, –Xs

C and C++ -specific Compiler Options

There are a large number of compiler options specific to the PGCC and PGC++ compilers, especially PGC++. This section provides the details of several of these options, but is not exhaustive. For a complete list of available options, including an exhaustive list of PGC++ options, use the –help command-line option. For further detail on a given option, use –help and specify the option explicitly, as described in –help .

–A

(pgcpp only) Instructs the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard, issuing errors for non-conforming code.

Default: By default, the compiler accepts code conforming to the standard C++ Annotated Reference Manual.

Usage: The following command-line requests ANSI conforming C++.

```
$ pgcpp -A hello.cc
```

Description: Use this option to instruct the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard and to issues errors for non-conforming code.

Related options:–a, –b and +p.

–a

(pgcpp only) Instructs the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard, issuing warnings for non-conforming code.

Default: By default, the compiler accepts code conforming to the standard C++ Annotated Reference Manual.

Usage: The following command-line requests ANSI conforming C++, issuing warnings for non-conforming code.

```
$ pgcpp -a hello.cc
```

Description: Use this option to instruct the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard and to issues warnings for non-conforming code.

Related options:–A, –b and +p.

–alias

select optimizations based on type-based pointer alias rules in C and C++.

Syntax:

```
-alias=[ansi|traditional]
```

Default: None.

Usage: The following command-line enables optimizations.

```
$ pgcpp -alias=ansi hello.cc
```

Description: Use this option to select optimizations based on type-based pointer alias rules in C and C++.

ansi
: Enable optimizations using ANSI C type-based pointer disambiguation

traditional
: Disable type-based pointer disambiguation

Related options: –Xt

--[no_]alternative_tokens

(pgcpp only) Enables or disables recognition of alternative tokens. These are tokens that make it possible to write C++ without the use of the comma (,) , [,], #, &, ^, and characters. The alternative tokens include the operator keywords (e.g., and, bitand, etc.) and digraphs.

Default:. The default behavior is --no_alternative_tokens, that is, to disable recognition of alternative tokens.

Usage: The following command-line enables alternative token recognition.

```
$ pgcpp --alternative_tokens hello.cc
```

(pgcpp only) Use this option to enable or disable recognition of alternative tokens. These tokens make it possible to write C++ without the use of the comma (,), [,], #, &, ^, and characters. The alternative tokens include digraphs and the operator keywords, such as *and*, *bitand*, and so on. The default behavior is disabled recognition of alternative tokens: --no_alternative_tokens.

Related options:

–B

(pgcc and pgcpp only) Enables use of C++ style comments starting with // in C program units.

Default: The PGCC ANSI and K&R C compiler does not allow C++ style comments.

Usage: In the following example the compiler accepts C++ style comments.

```
$ pgcc -B myprog.cc
```

Description: Use this option to enable use of C++ style comments starting with // in C program units.

Related options: –Mcpp

–b

(pgcpp only) Enables compilation of C++ with cfront 2.1 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example the compiler accepts cfront constructs.

```
$ pgcpp -b myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 2.1 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 2.1).

This option also enables acceptance of anachronisms.

Related options: —cfront2.1, –b3 , —cfront3.0, +p, –A

–b3

(pgcpp only) Enables compilation of C++ with cfront 3.0 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example, the compiler accepts cfront constructs.

```
$ pgcpp -b3 myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 3.0 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 3.0).

This option also enables acceptance of anachronisms.

Related options: —cfront2.1, –b, —cfront3.0, +p, –A

--[no_]bool

(pgcpp only) Enables or disables recognition of bool.

Default: The compile recognizes bool: --bool.

Usage: In the following example, the compiler does not recognize bool.

```
$ pgcpp --no_bool myprog.cc
```

Description: Use this option to enable or disable recognition of bool.

Related options: None.

--[no_]builtin

Compile with or without math subroutine builtin support.

Default: The default is to compile with math subroutine support: --builtin.

Usage: In the following example, the compiler does not build with math subroutine support.

```
$ pgcpp --no_builtin myprog.cc
```

Description: Use this option to enable or disable compiling with math subroutine builtin support. When you compile with math subroutine builtin support, the selected math library routines are inlined.

Related options:

--cfront_2.1

(pgcpp only) Enables compilation of C++ with cfront 2.1 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example, the compiler accepts cfront constructs.

```
$ pgcpp --cfront_2.1 myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 2.1 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 2.1).

This option also enables acceptance of anachronisms.

Related options: –b, –b3, —cfront3.0, +p, –A

--cfront_3.0

(pgcpp only) Enables compilation of C++ with cfront 3.0 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example, the compiler accepts cfront constructs.

```
$ pgcpp --cfront_3.0 myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 3.0 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 3.0).

This option also enables acceptance of anachronisms.

Related options: --cfront2.1, –b, –b3, +p, –A

--compress_names

Compresses long function names in the file.

Default: The compiler does not compress names: --no_compress_names.

Usage: In the following example, the compiler compresses long function names.

```
$ pgcpp --ccmpress_names myprog.cc
```

Description: Use this option to specify to compress long function names. Highly nested template parameters can cause very long function names. These long names can cause problems for older assemblers. Users encountering these problems should compile all C++ code, including library code with the switch `--compress_names`. Libraries supplied by PGI work with --compress_names.

Related options: None.

--create_pch filename

(pgcpp only) If other conditions are satisfied, create a precompiled header file with the specified name.

Note

If --pch (automatic PCH mode) appears on the command line following this option, its effect is erased.

Default: The compiler does not create a precompiled header file.

Usage: In the following example, the compiler creates a precompiled header file, `hdr1`.

```
$ pgcpp --create_pch hdr1 myprog.cc
```

Description: If other conditions are satisfied, use this option to create a precompiled header file with the specified name.

Related options: --pch

--diag_error tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options:--diag_remark tag, --diag_suppress tag, --diag_warning tag, --display_error_number

--diag_remark tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options: --diag_error tag, --diag_suppress tag, --diag_warning tag, --display_error_number

--diag_suppress tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Usage: In the following example, the compiler overrides the normal error severity of the specified diagnostic messages.

```
$ pgcpp --diag_suppress error_tag prog.cc
```

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options:--diag_error tag, --diag_remark tag, --diag_warning tag, --diag_error_number

--diag_warning tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Usage: In the following example, the compiler overrides the normal error severity of the specified diagnostic messages.

```
$ pgcpp --diag_suppress an_error_tag myprog.cc
```

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options: --diag_error tag, --diag_remark tag, --diag_suppress tag, --diag_error_number

--display_error_number

(pgcpp only) Displays the error message number in any diagnostic messages that are generated. The option may be used to determine the error number to be used when overriding the severity of a diagnostic message.

Default: The compiler does not display error message numbers for generated diagnostic messages.

Usage: In the following example, the compiler displays the error message number for any generated diagnostic messages.PLEASE PROVIDE ONE

```
$ pgcpp --display_error_number myprog.cc
```

Description: Use this option to display the error message number in any diagnostic messages that are generated. You can use this option to determine the error number to be used when overriding the severity of a diagnostic message.

Related options: --diag_error tag, --diag_remark tag, --diag_suppress tag, --diag_warning tag

-e<number>

(pgcpp only) Set the C++ front-end error limit to the specified <number>.

--[no_]exceptions

(pgcpp only) Enables or disables exception handling support.

Default: The compiler provides exception handling support: --exceptions.

Usage: In the following example, the compiler does not provide exception handling support. PLEASE PROVIDE ONE

```
$ pgcpp --no_exceptions myprog.cc
```

Description: Use this option to enable or disable exception handling support.

Related options:--zc_eh

--gnu_extensions

(pgcpp only) Allows GNU extensions.

Default: The compiler does not allow GNU extensions.

Usage: In the following example, the compiler allows GNU extensions.

```
$ pgcpp --gnu_extensions myprog.cc
```

Description: Use this option to allow GNU extensions, such as "include next", which are required to compile Linux system header files.

Related options:--zc_eh

--[no]llalign

(pgcpp only) Enables or disables alignment of long long integers on long long boundaries.

Default: The compiler aligns long long integers on long long boundaries: --llalign.

Usage: In the following example, the compiler does not align long long integers on long long boundaries.

```
$ pgcpp --nollalign myprog.cc
```

Description: Use this option to allow enable or disable alignment of long long integers on long long boundaries.

Related options: –Mipa=[no]align

–M

Generates a list of make dependencies and prints them to stdout.

Note

The compilation stops after the preprocessing phase.

Default: The compiler does not generate a list of make dependencies.

Usage: In the following example, the compiler generates a list of make dependencies.

```
$ pgcpp -M myprog.cc
```

Description: Use this option to generate a list of make dependencies and prints them to stdout.

Related options:–MD, –P

–MD

Generates a list of make dependencies and prints them to a file.

Default: The compiler does not generate a list of make dependencies.

Usage: In the following example, the compiler generates a list of make dependencies and prints them to the file myprog.d.

```
$ pgcpp -MD myprog.cc
```

Description: Use this option to generate a list of make dependencies and prints them to a file. The name of the file is determined by the name of the file under compilation.dependencies_file<file>.

Related options:–M, –P

--optk_allow_dollar_in_id_chars

(pgcpp only) Accepts dollar signs ($) in identifiers.

Default: The compiler does not accept dollar signs ($) in identifiers.

Usage: In the following example, the compiler allows dollar signs ($) in identifiers.

```
$ pgcpp -optk_allow_dollar_in_id_chars myprog.cc
```

Description: Use this option to instruct the compiler to accept dollar signs ($) in identifiers.

–P

Halts the compilation process after preprocessing and writes the preprocessed output to a file.

Default: The compiler produces an executable file.

Usage: In the following example, the compiler produces the preprocessed file myprog.i in the current directory.

```
$ pgcpp -P myprog.cc
```

Description: Use this option to halt the compilation process after preprocessing and write the preprocessed output to a file. If the input file is `filename.c` or `filename.cc.`, then the output file is `filename.i`.

Related options: –C,–c,–E, –Mkeepasm, –o, –S

-+p

(pgcpp only) Disallow all anachronistic constructs.

Default: The compiler disallows all anachronistic constructs.

Usage: In the following example, the compiler disallows all anachronistic constructs.

```
$ pgcpp -+p myprog.cc
```

Description: Use this option to disallow all anachronistic constructs.

Related options: None.

--pch

(pgcpp only) Automatically use and/or create a precompiled header file.

Note

If --use_pch or --create_pch (manual PCH mode) appears on the command line following this option, this option has no effect.

Default: The compiler does not automatically use or create a precompiled header file.

Usage: In the following example, the compiler automatically uses a precompiled header file.

```
$ pgcpp --pch myprog.cc
```

Description: Use this option to automatically use and/or create a precompiled header file.

Related options:--create_pch, --pc_dir, --use_pch

--pch_dir directoryname

(pgcpp only) Specifies the directory in which to search for and/or create a precompiled header file.

The compiler searches your PATH for precompiled header files / use or create a precompiled header file.

Usage: In the following example, the compiler searches in the directory `myhdrdir` for a precompiled header file.

```
$ pgcpp --pch_dir myhdrdir myprog.cc
```

Description: Use this option to specify the directory in which to search for and/or create a precompiled header file. You may use this option with automatic PCH mode (--pch) or manual PCH mode (--create_pch or --use_pch).

Related options:--create_pch, --pch, --use_pch

--[no_]pch_messages

(pgcpp only) Enables or disables the display of a message indicating that the current compilation used or created a precompiled header file.

The compiler displays a message when it uses or creates a precompiled header file.

In the following example, no message is displayed when the precompiled header file located in `myhdrdir` is used in the compilation.

```
$ pgcpp --pch_dir myhdrdir --no_pch_messages myprog.cc
```

Description: Use this option to enable or disable the display of a message indicating that the current compilation used or created a precompiled header file.

Related options:--pch_dir

--preinclude=<filename>

(pgcpp only) Specifies the name of a file to be included at the beginning of the compilation.

In the following example, the compiler includes the file `incl_file.c` at the beginning of the compilation. me

```
$ pgcpp --preinclude=incl_file.c myprog.cc
```

Description: Use this option to specify the name of a file to be included at the beginning of the compilation. For example, you can use this option to set system-dependent macros and types.

Related options: None.

--use_pch filename

(pgcpp only) Uses a precompiled header file of the specified name as part of the current compilation.

Note

If --pch (automatic PCH mode) appears on the command line following this option, its effect is erased.

Default: The compiler does not use a precompiled header file.

In the following example, the compiler uses the precompiled header file, `hdr1` as part of the current compilation.

```
$ pgcpp --use_pch hdr1 myprog.cc
```

Use a precompiled header file of the specified name as part of the current compilation. If --pch (automatic PCH mode) appears on the command line following this option, its effect is erased.

Related options:--create_pch, --pch_dir, --pch_messages

--[no_]using_std

(pgcpp only) Enables or disables implicit use of the std namespace when standard header files are included.

Default: The compiler uses std namespace when standard header files are included: --using_std.

Usage: The following command-line disables implicit use of the std namespace:

```
$ pgcpp --no_using_std hello.cc
```

Description: Use this option to enable or disable implicit use of the std namespace when standard header files are included in the compilation.

Related options: –M[no]stddef

–t

(pgcpp only) Control instantiation of template functions.

–t [arg]

Default: No templates are instantiated.

Usage: In the following example, all templates are instantiated.

```
$ pgcpp -tall myprog.cc
```

Description: Use this option to control instantiation of template functions. The argument is one of the following:

all
: Instantiates all functions whether or not they are used.

local
: Instantiates only the functions that are used in this compilation, and forces those functions to be local to this compilation.

  Note: This may cause multiple copies of local static variables. If this occurs, the program may not execute correctly.

none
: Instantiates no functions. This is the default.

used
: Instantiates only the functions that are used in this compilation.

Usage: In the following example, all templates are instantiated.

```
 $ pgcpp
-tall myprog.cc
```

–X

(pgcpp only) Generates cross-reference information and places output in the specified file.

Syntax:

–Xfoo

where foo is the specifies file for the cross reference information.

Default: The compiler does not generate cross-reference information.

Usage: In the following example, the compiler generates cross-reference information, placing it in the file: `xreffile`.

```
$ pgcpp -Xxreffile myprog.cc
```

Description: Use this option to generate cross-reference information and place output in the specified file. This is an EDG option.

Related options: None.

--zc_eh

(Linux only) Generates zero-overhead exceptionregions.

Default: The compiler does not use --zc_eh but instead uses --sjlj_eh, which implements exception handling with setjmp and longjmp.

Usage: The following command-line enables zero-overhead exception regions:

```
$ pgcpp --zc_eh ello.cc
```

Description: Use this option to generate zero-overhead exception regions. The --zc_eh option defers the cost of exception handling until an exception is thrown. For a program with many exception regions and few throws, this option may lead to improved run-time performance.

This option is compatible with C++ code that was compiled with previous version if PGI C++.

Note

The --zc_eh option is available only on newer Linux systems that supply the system unwind libraries in libgcc_eh and on Windows.

Related options: --[no]exceptions.

–M Options by Category

This section describes each of the options available with –M by the categories:

Code generation	Fortran Language Controls	Optimization	Environment
C/C++ Language Controls	Inlining	Miscellaneous	

For a complete alphabetical list of all the options, refer to “ –M Options Summary,” on page 218.

The following sections provide detailed descriptions of several, but not all, of the –M<pgflag> options. For a complete alphabetical list of all the options, refer to “ –M Options Summary,” on page 218. These options are grouped according to categories and are listed with exact syntax, defaults, and notes concerning similar or

related options. For the latest information and description of a given option, or to see all available options, use the –help command-line option, described in"–help ," on page 212.

Code Generation Controls

This section describes the –M<pgflag> options that control code generation.

Default: For arguments that you do not specify, the default code generation controls are these:

nodaz	norecursive	nosecond_underscore
noflushz	noreentrant	nostride0
largeaddressaware	noref_externals	signextend

Related options: –D, –I, –L, –l, –U

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Mdaz
: Set IEEE denormalized input values to zero; there is a performance benefit but misleading results can occur, such as when dividing a small normalized number by a denormalized number. To take effect, this option must be set for the main program.

–Mnodaz
: Do not treat denormalized numbers as zero.To take effect, this option must be set for the main program.

–Mnodwarf
: Specifies not to add DWARF debug information; must be used in combination with `-g`.

–Mdwarf1
: Generate DWARF1 format debug information; must be used in combination with `-g`.

–Mdwarf2
: Generate DWARF2 format debug information; must be used in combination with `-g`.

–Mdwarf3
: Generate DWARF3 format debug information; must be used in combination with `-g`.

–Mflushz
: Set SSE flush-to-zero mode; if a floating-point underflow occurs, the value is set to zero.To take effect, this option must be set for the main program.

–Mnoflushz
: Do not set SSE flush-to-zero mode; generate underflows.To take effect, this option must be set for the main program.

–Mfunc32
: Align functions on 32-byte boundaries.

–Minstrument[=functions] linx86-64 only
: Generate additional code to enable instrumentation of functions. The option `-Minstrument=functions` is the same as `-Minstrument`.

Implies `-Minfo=ccff` and `-Mframe`.

`-Mlargeaddressaware=[no]`

[Win64 only] Generates code that allows for addresses greater than 2GB, using RIP-relative addressing.

Use `-Mlargeaddressaware=no` for a direct addressing mechanism that restricts the total addressable memory.

> **Note**
>
> Do not use `-Mlargeaddressaware=no` if the object file will be placed in a DLL.

If `-Mlargeaddressaware=no` is used to compile any object file, it must also be used when linking.

`-Mlarge_arrays`

Enable support for 64-bit indexing and single static data objects larger than 2GB in size. This option is default in the presence of `-mcmodel=medium`. Can be used separately together with the default small memory model for certain 64-bit applications that manage their own memory space. For more information, refer to Chapter 13, "*Programming Considerations for 64-Bit Environments*".

`-Mmpi=option`

`-Mmpi` adds the include and library options to the compile and link commands necessary to build an MPI application using MPI header files and libraries.

To use `-Mmpi`, you must have a version of MPI installed on your system.

On Linux, this option inserts `-I$MPIDIR/include` into the compile line and `-L$MPIDIR/lib` into the link line. The specifies option determines whether to select MPICH-1 or MPICH-2 headers and libraries. The base directories for MPICH-1 and MPICH-2 are set in `localrc`.

On Windows, PGI compilers and tools support Microsoft's implementation of MPI, MSMPI. This version of MPI is available with Microsoft's HPC Pack 2008 SDK.

The -Mmpi options are as specified:

- –Mmpi=hpmpi - (Linux only) Select the HP-MPI communication libraries and associated header files if they are installed.
- –Mmpi=mpich1 - Selects preconfigured MPICH-1 communication libraries.
- –Mmpi=mpich2 - Selects preconfigured MPICH-2 communication libraries.
- –Mmpi=msmpi - Select Microsoft MSMPI libraries.
- –Mmpi=mvapich1 - (Linux only) Selects default MVAPICH communication libraries that are available only from the PGI Cluster Development Kit

For more information on these options, refer to Chapter 6, "*Using MPI* ," on page 73.

> **Note**
>
> The user can set the environment variables `MPIDIR` and `MPILIBNAME` to override the default locations for the MPI directory and library name.

On Windows, the user can set the appropriate environment variable, either `CCP_HOME` or `CCP_SDK` to override the default location for the directory associated with using MSMPI.

For `-Mmpi=msmpi` to work, the `CCP_HOME` environment variable must be set. When the Microsoft HPC Pack 2008 SDK is installed, this variable is typically set to point to the MSMPI library directory.

`–Mnolarge_arrays`
Disable support for 64-bit indexing and single static data objects larger than 2GB in size. When placed after `-mcmodel=medium` on the command line, disables use of 64-bit indexing for applications that have no single data object larger than 2GB.

`–Mnomain`
Instructs the compiler not to include the object file that calls the Fortran main program as part of the link step. This option is useful for linking programs in which the main program is written in C/C++ and one or more subroutines are written in Fortran (pgf77, pgf95, and pghpf only).

`–M[no]movnt`
Instructs the compiler to generate nontemporal move and prefetch instructions even in cases where the compiler cannot determine statically at compile-time that these instructions will be beneficial.

`–Mpre`
enables partial redundancy elimination.

`–Mprof[=option[,option,...]]`
Set performance profiling options. Use of these options causes the resulting executable to create a performance profile that can be viewed and analyzed with the PGPROF performance profiler. In the descriptions that follow, PGI-style profiling implies compiler-generated source instrumentation. MPICH-style profiling implies the use of instrumented wrappers for MPI library routines.

The option argument can be any of the following:

`[no]ccff`
Enable [disable] common compiler feedback format, CCFF, information.

`dwarf`
Generate limited DWARF symbol information sufficient for most performance profilers.

`func`
Perform PGI-style function-level profiling.

`hpmpi`
Use the profiled HPMPI communication library. Implies `-Mpfi=hpmpi`. For more information, refer to "Using HP-MPI on Linux".

`hwcts`
Generate a profile using event-based sampling of hardware counters via the PAPI interface. (linux86-64 platform only; PAPI must be installed).

`lines`
Perform PGI-style line-level profiling.

`mpich1`
Perform MPICH-style profiling for MPICH-1. Implies `-Mmpi=mpich1`. (Linux only).

`mpich2`
Perform MPICH-style profiling for MPICH-2. Implies `-Mmpi=mpich2`. (Linux with MPI support licence privileges only.) For more information, refer to "Using MPICH-2 on Linux".

`msmpi`
Perform MPICH-style profiling for Microsoft MPI. Implies `-Mmpi=msmpi`. (This option is valid only if Microsoft HPC Pack 2008 SDK is installed.) For more information, refer to "Using MSMPI on Windows".

`mvapich1`
Use profiled MVAPICH communication library. Implies `-Mmpi=mvapich1`. (Linux only). For or more information, refer to "Using MVAPICH on Linux".

`time`
Generate a profile using time-based instruction-level statistical sampling. This is equivalent to `-pg`, except that the profile is saved to a file names `pgprof.out` rather than `gmon.out`.

`-Mrecursive`
instructs the compiler to allow Fortran subprograms to be called recursively.

`-Mnorecursive`
Fortran subprograms may not be called recursively.

`-Mref_externals`
force references to names appearing in EXTERNAL statements (pgf77, pgf95, and pghpf only).

`-Mnoref_externals`
do not force references to names appearing in EXTERNAL statements (pgf77, pgf95, and pghpf only).

`-Mreentrant`
instructs the compiler to avoid optimizations that can prevent code from being reentrant.

`-Mnoreentrant`
instructs the compiler not to avoid optimizations that can prevent code from being reentrant.

`-Msecond_underscore`
instructs the compiler to add a second underscore to the name of a Fortran global symbol if its name already contains an underscore. This option is useful for maintaining compatibility with object code compiled using g77, which uses this convention by default (pgf77, pgf95, and pghpf only).

`-Mnosecond_underscore`
instructs the compiler not to add a second underscore to the name of a Fortran global symbol if its name already contains an underscore (pgf77, pgf95, and pghpf only).

`-Msignextend`
instructs the compiler to extend the sign bit that is set as a result of converting an object of one data type to an object of a larger signed data type.

`-Mnosignextend`
instructs the compiler not to extend the sign bit that is set as the result of converting an object of one data type to an object of a larger data type.

–Msafe_lastval
When a scalar is used after a loop, but is not defined on every iteration of the loop, the compiler does not by default parallelize the loop. However, this option tells the compiler it's safe to parallelize the loop. For a given loop the last value computed for all scalars makes it safe to parallelize the loop.

–Mstride0
instructs the compiler to inhibit certain optimizations and to allow for stride 0 array references. This option may degrade performance and should only be used if zero-stride induction variables are possible.

–Mnostride0
instructs the compiler to perform certain optimizations and to disallow for stride 0 array references.

–Munix
use UNIX symbol and parameter passing conventions for Fortran subprograms (pgf77, pgf95, and pghpf for Win32 only).

–Mvarargs
force Fortran program units to assume procedure calls are to C functions with a varargs-type interface (pgf77 and pgf95 only).

C/C++ Language Controls

This section describes the –M<pgflag> options that affect C/C++ language interpretations by the PGI C and C++ compilers. These options are only valid to the pgcc and pgcpp compiler drivers.

Default: For arguments that you do not specify, the defaults are as follows:

noasmkeyword	nosingle
dollar,_	schar

Usage:

In this example, the compiler allows the asm keyword in the source file.

```
$ pgcc -Masmkeyword myprog.c
```

In the following example, the compiler maps the dollar sign to the dot character.

```
$ pgcc -Mdollar,. myprog.c
```

In the following example, the compiler treats floating-point constants as float values.

```
$ pgcc -Mfcon myprog.c
```

In the following example, the compiler does not convert float parameters to double parameters.

```
$ pgcc -Msingle myprog.c
```

Without –Muchar or with –Mschar, the variable ch is a signed character:

```
char ch;
signed char sch;
```

If –Muchar is specified on the command line:

```
$ pgcc -Muchar myprog.c
```

char ch in the preceding declaration is equivalent to:

```
unsigned char ch;
```

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Masmkeyword
: instructs the compiler to allow the asm keyword in C source files. The syntax of the asm statement is as follows:

  ```
  asm("statement");
  ```

  Where statement is a legal assembly-language statement. The quote marks are required.

  Note. The current default is to support gcc's extended asm, where the syntax of extended asm includes asm strings. The –M[no]asmkeyword switch is useful only if the target device is a Pentium 3 or older cpu type (–tp piii|p6|k7|athlon|athlonxp|px).

–Mnoasmkeyword
: instructs the compiler not to allow the asm keyword in C source files. If you use this option and your program includes the asm keyword, unresolved references will be generated

–Mdollar,char
: char specifies the character to which the compiler maps the dollar sign ($). The PGCC compiler allows the dollar sign in names; ANSI C does not allow the dollar sign in names.

–Mfcon
: instructs the compiler to treat floating-point constants as float data types, instead of double data types. This option can improve the performance of single-precision code.

–M[no]m128
: instructs the compiler to recognize [ignore] __m128, __m128d, and __m128i datatypes. floating-point constants as float data types, instead of double data types. This option can improve the performance of single-precision code.

–Mschar
: specifies signed char characters. The compiler treats "plain" char declarations as signed char.

–Msingle
: do not to convert float parameters to double parameters in non-prototyped functions. This option can result in faster code if your program uses only float parameters. However, since ANSI C specifies that routines must convert float parameters to double parameters in non-prototyped functions, this option results in non#ANSI conformant code.

–Mnosingle
: instructs the compiler to convert float parameters to double parameters in non-prototyped functions.

–Muchar
: instructs the compiler to treat "plain" char declarations as unsigned char.

Environment Controls

This section describes the –M<pgflag> options that control environments.

Default: For arguments that you do not specify, the default environment option depends on your configuration.

The following list provides the syntax for each –M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Mlfs
(32-bit Linux only) link in libraries that enable file I/O to files larger than 2GB (Large File Support).

–Mnostartup
instructs the linker not to link in the standard startup routine that contains the entry point (_start) for the program.

Note

If you use the –Mnostartup option and do not supply an entry point, the linker issues the following error message: Warning: cannot find entry symbol _start

–M[no]smartalloc[=huge|h[uge:<n>|hugebss|nohuge]
adds a call to the routine mallopt in the main routine. This option supports large TLBs on Linux and Windows. This option must be used to compile the main routine to enable optimized malloc routines.

The option arguments can be any of the following:

huge
Link in the huge page runtime library.

Enables large 2-megabyte pages to be allocated. The effect is to reduce the number of TLB entries required to execute a program. This option is most effective on Barcelona and Core 2 systems; older architectures do not have enough TLB entries for this option to be beneficial. By itself, the huge suboption tries to allocate as many huge pages as required.

huge:<n>
Link the huge page runtime library and allocate n huge pages. Use this suboption to limit the number of huge pages allocated to n.

You can also limit the pages allocated by using the environment variable PGI_HUGE_PAGES.

hugebss
(64-bit only) Puts the BSS section in huge pages; attempts to put a program's uninitialized data section into huge pages.

Note

This flag dynamically links the library libhugetlbfs_pgi even if –Bstatic is used.

nohuge
Overrides a previous –Msmartalloc=huge setting.

Tip

To be effective, this switch must be specified when compiling the file containing the Fortran, C, or C++ main program.

–M[no]stddef
instructs the compiler not to predefine any macros to the preprocessor when compiling a C program.

–Mnostdinc
instructs the compiler to not search the standard location for include files.

–Mnostdlib
instructs the linker not to link in the standard libraries libpgftnrtl.a, libm.a, libc.a, and libpgc.a in the library directory lib within the standard directory. You can link in your own library with the –l option or specify a library directory with the –L option.

Fortran Language Controls

This section describes the –M<pgflag> options that affect Fortran language interpretations by the PGI Fortran compilers. These options are valid only for the pghpf, pgf77 and pgf95 compiler drivers.

Default: For arguments that you do not specify, the defaults are as follows:

nobackslash	nodefaultunit	dollar,_	noonetrip	nounixlogical
nodclchk	nodlines	noiomutex	nosave	noupcase

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Mallocatable=95|03
controls whether Fortran 95 or Fortran 2003 semantics are used in allocatable array assignments. The default behavior is to use Fortran 95 semantics; the 03 option instructs the compiler to use Fortran 2003 semantics.

–Mbackslash
the compiler treats the backslash as a normal character, and not as an escape character in quoted strings.

–Mnobackslash
the compiler recognizes a backslash as an escape character in quoted strings (in accordance with standard C usage).

–Mdclchk
the compiler requires that all program variables be declared.

–Mnodclchk
the compiler does not require that all program variables be declared.

–Mdefaultunit
the compiler treats "*" as a synonym for standard input for reading and standard output for writing.

–Mnodefaultunit
the compiler treats "*" as a synonym for unit 5 on input and unit 6 on output.

–Mdlines
the compiler treats lines containing "D" in column 1 as executable statements (ignoring the "D").

–Mnodlines
the compiler does not treat lines containing "D" in column 1 as executable statements (does not ignore the "D").

–Mdollar,char
char specifies the character to which the compiler maps the dollar sign. The compiler allows the dollar sign in names.

–Mextend
the compiler accepts 132-column source code; otherwise it accepts 72-column code.

–Mfixed
the compiler assumes input source files are in FORTRAN 77-style fixed form format.

–Mfree
the compiler assumes the input source files are in Fortran 90/95 freeform format.

–Miomutex
the compiler generates critical section calls around Fortran I/O statements.

–Mnoiomutex
the compiler does not generate critical section calls around Fortran I/O statements.

–Monetrip
the compiler forces each DO loop to execute at least once.

–Mnoonetrip
the compiler does not force each DO loop to execute at least once. This option is useful for programs written for earlier versions of Fortran.

–Msave
the compiler assumes that all local variables are subject to the SAVE statement. Note that this may allow older Fortran programs to run, but it can greatly reduce performance.

–Mnosave
the compiler does not assume that all local variables are subject to the SAVE statement.

–Mstandard
the compiler flags non-ANSI-conforming source code.

–Munixlogical
directs the compiler to treat logical values as true if the value is non-zero and false if the value is zero (UNIX F77 convention.) When –Munixlogical is enabled, a logical value or test that is non-zero is .TRUE., and a value or test that is zero is .FALSE.. In addition, the value of a logical expression is guaranteed to be one (1) when the result is .TRUE..

–Mnounixlogical
directs the compiler to use the VMS convention for logical values for true and false. Even values are true and odd values are false.

–Mupcase
the compiler preserves uppercase letters in identifiers. With –Mupcase, the identifiers "X" and "x" are different. Keywords must be in lower case. This selection affects the linking process. If you compile and

link the same source code using `-Mupcase` on one occasion and `-Mnoupcase` on another, you may get two different executables - depending on whether the source contains uppercase letters. The standard libraries are compiled using the default `-Mnoupcase` .

–Mnoupcase
: the compiler converts all identifiers to lower case. This selection affects the linking process: If you compile and link the same source code using –Mupcase on one occasion and –Mnoupcase on another, you may get two different executables (depending on whether the source contains uppercase letters). The standard libraries are compiled using –Mnoupcase.

Inlining Controls

This section describes the –M<pgflag> options that control function inlining. Before looking at all the options, let's look at a couple examples.

Usage: In the following example, the compiler extracts functions that have 500 or fewer statements from the source file `myprog.f` and saves them in the file `extract.il`.

```
$ pgf95 -Mextract=500 -o extract.il myprog.f
```

In the following example, the compiler inlines functions with fewer than approximately 100 statements in the source file `myprog.f`.

```
$ pgf95 -Minline=size:100 myprog.f
```

Related options: –o, –Mextract

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–M[no]autoinline
: instructs the compiler to inline a C/C++ function at –O2 and when it is declared with the inline keyword.

–Mextract[=option[,option,...]]

: Extracts functions from the file indicated on the command line and creates or appends to the specified extract directory where option can be any of:

  name:func
  : instructs the extractor to extract function func from the file.

  size:number
  : instructs the extractor to extract functions with number or fewer statements from the file.

  lib:filename.ext
  : Use directory filename.ext as the extract directory (required in order to save and re-use inline libraries).

  If you specify both name and size, the compiler extracts functions that match func, or that have number or fewer statements. For examples of extracting functions, see Chapter 4, "*Using Function Inlining*".

–Minline[=option[,option,...]]
: This passes options to the function inliner, where the option can be any of these:

except:func
: instructs the inliner to inline all eligible functions except func, a function in the source text. Multiple functions can be listed, comma-separated.

[name:]func
: instructs the inliner to inline the function func. The func name should be a non-numeric string that does not contain a period. You can also use a name: prefix followed by the function name. If name: is specified, what follows is always the name of a function.

[lib:]filename.ext
: instructs the inliner to inline the functions within the library file `filename.ext`. The compiler assumes that a `filename.ext` option containing a period is a library file. Create the library file using the `–Mextract` option. You can also use a lib: prefix followed by the library name. If lib: is specified, no period is necessary in the library name. Functions from the specified library are inlined. If no library is specified, functions are extracted from a temporary library created during an extract prepass.

levels:number
: instructs the inliner to perform number levels of inlining. The default number is 1.

[no]reshape
: instructs the inliner to allow (disallow) inlining in Fortran even when array shapes do not match. The default is `–Minline=noreshape`, except with `–Mconcur` or `–mp`, where the default is `–Minline=reshape`,=reshape.

[size:]number
: instructs the inliner to inline functions with number or fewer statements. You can also use a size: prefix followed by a number. If size: is specified, what follows is always taken as a number.

If you specify both func and number, the compiler inlines functions that match the function name or have number or fewer statements. For examples of inlining functions, refer to Chapter 4, "*Using Function Inlining*".

Optimization Controls

This section describes the –M<pgflag> options that control optimization. Before looking at all the options, let's look at the defaults.

Default: For arguments that you do not specify, the default optimization control options are as follows:

depchk	noipa	nounroll	nor8
i4	nolre	novect	nor8intrinsics
nofprelaxed	noprefetch		

Note

If you do not supply an option to `–Mvect`, the compiler uses defaults that are dependent upon the target system.

Usage: In this example, the compiler invokes the vectorizer with use of packed SSE instructions enabled.

```
$ pgf95 -Mvect=sse -Mcache_align myprog.f
```

Related options: –g, –O

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Mcache_align
: Align unconstrained objects of length greater than or equal to 16 bytes on cache-line boundaries. An unconstrained object is a data object that is not a member of an aggregate structure or common block. This option does not affect the alignment of allocatable or automatic arrays.

  Note

  To effect cache-line alignment of stack-based local variables, the main program or function must be compiled with –Mcache_align.

–Mconcur[=option [,option,...]]
: Instructs the compiler to enable auto-concurrentization of loops. If –Mconcur is specified, multiple processors will be used to execute loops that the compiler determines to be parallelizable. Where option is one of the following:

  allcores
  : Instructs the compiler to use all available cores. Use this option at link time.

  [no]altcode:n
  : Instructs the parallelizer to generate alternate serial code for parallelized loops. If altcode is specified without arguments, the parallelizer determines an appropriate cutoff length and generates serial code to be executed whenever the loop count is less than or equal to that length. If altcode:n is specified, the serial altcode is executed whenever the loop count is less than or equal to n. If noaltcode is specified, the parallelized version of the loop is always executed regardless of the loop count.

  bind
  : Instructs the parallelizer to bind threads to cores. Use this option at link time.

  cncall
  : Calls in parallel loops are safe to parallelize. Loops containing calls are candidates for parallelization. Also, no minimum loop count threshold must be satisfied before parallelization will occur, and last values of scalars are assumed to be safe.

  dist:block
  : Parallelize with block distribution (this is the default). Contiguous blocks of iterations of a parallelizable loop are assigned to the available processors.

  dist:cyclic
  : Parallelize with cyclic distribution. The outermost parallelizable loop in any loop nest is parallelized. If a parallelized loop is innermost, its iterations are allocated to processors cyclically. For example, if there are 3 processors executing a loop, processor 0 performs iterations 0, 3, 6, etc.; processor 1 performs iterations 1, 4, 7, etc.; and processor 2 performs iterations 2, 5, 8, etc.

  [no]innermost
  : Enable parallelization of innermost loops. The default is to not parallelize innermost loops, since it is usually not profitable on dual-core processors.

noassoc
: Disables parallelization of loops with reductions.

When linking, the `-Mconcur` switch must be specified or unresolved references will result. The `NCPUS` environment variable controls how many processors or cores are used to execute parallelized loops.

> **Note**
>
> This option applies only on shared-memory multi-processor (SMP) or multi-core processor-based systems.

–`Mcray[=option[,option,...]]`
: (pgf77 and pgf95 only) Force Cray Fortran (CF77) compatibility with respect to the listed options. Possible values of option include:

pointer
: for purposes of optimization, it is assumed that pointer-based variables do not overlay the storage of any other variable.

–`Mdepchk`
: instructs the compiler to assume unresolved data dependencies actually conflict.

–`Mnodepchk`
: Instructs the compiler to assume potential data dependencies do not conflict. However, if data dependencies exist, this option can produce incorrect code.

–`Mdse`
: Enables a dead store elimination phase that is useful for programs that rely on extensive use of inline function calls for performance. This is disabled by default.

–`Mnodse`
: Disables the dead store elimination phase. This is the default.

–`M[no]fpapprox [=option]`
: Perform certain fp operations using low-precision approximation.

`-Mnofpapprox` specifies not to use low-precision fp approximation operations.

By default `-Mfpapprox` is not used.

If `-Mfpapprox` is used without suboptions, it defaults to use approximate `div`, `sqrt`, and `rsqrt`. The available suboptions are these:

div
: Approximate floating point division

sqrt
: Approximate floating point square root

rsqrt
: Approximate floating point reciprocal square root

–M[no]fpmisalign
Instructs the compiler to allow (not allow) vector arithmetic instructions with memory operands that are not aligned on 16-byte boundaries. The default is –Mnofpamisalign on all processors.

Note

Applicable only with one of these options: –tp barcelona or –tp barcelona-64

–Mf[=option]
Instructs the compiler to use relaxed precision in the calculation of some intrinsic functions. Can result in improved performance at the expense of numerical accuracy.

The possible values for option are:

div
Perform divide using relaxed precision.

noorder
Perform reciprocal square root (1/sqrt) using relaxed precision.

order
Perform reciprocal square root (1/sqrt) using relaxed precision.

recip
Perform reciprocal using relaxed precision.

rsqrt
Perform reciprocal square root (1/sqrt) using relaxed precision.

sqrt
Perform square root with relaxed precision.

With no options, –Mfprelaxed generates relaxed precision code for those operations that generate a significant performance improvement, depending on the target processor.

–Mnofprelaxed
(default) instructs the compiler to not use relaxed precision in the calculation of intrinsic functions.

–Mi4
(pgf77 and pgf95 only) the compiler treats INTEGER variables as INTEGER*4.

–Mipa=<option>[,<option>[,...]]
Pass options to the interprocedural analyzer.

Note

–Mipa implies –O2, and the minimum optimization level that can be specified in combination with –Mipa is –O2.

For example, if you specify –Mipa –O1 on the command line, the optimization level is automatically elevated to –O2 by the compiler driver. Typically, as recommended, you would use –Mipa=fast.

Many of the following sub-options can be prefaced with no, which reverses or disables the effect of the sub-option if it's included in an aggregate sub-option such as –Mipa=fast. The choices of option are:

[no]align
: recognize when targets of a pointer dummy are aligned. The default is noalign.

[no]arg
: remove arguments replaced by const, ptr. The default is noarg.

[no]cg
: generate call graph information for viewing using the `pgicg` command-line utility. The default is nocg.

[no]const
: perform interprocedural constant propagation. The default is const.

except:<func>
: used with inline to specify functions which should not be inlined. The default is to inline all eligible functions according to internally defined heuristics. Valid only immediately following the inline suboption.

[no]f90ptr
: F90/F95 pointer disambiguation across calls. The default is nof90ptr.

fast
: choose IPA options generally optimal for the target. To see settings for `–Mipa=fast` on a given target, use `–help`.

force
: force all objects to re-compile regardless of whether IPA information has changed.

[no]globals
: optimize references to global variables. The default is noglobals.

inline[:n]
: perform automatic function inlining. If the optional :n is provided, limit inlining to at most n levels. IPA-based function inlining is performed from leaf routines upward.

ipofile
: save IPA information in an `.ipo` file rather than incorporating it into the object file.

jobs[:n]
: recompile n jobs in parallel and print source file names as they are compiled.

[no]keepobj
: keep the optimized object files, using file name mangling, to reduce re-compile time in subsequent builds. The default is keepobj.

[no]libc
: optimize calls to certain standard C library routines. The default is nolibc.

[no]libinline
: allow inlining of routines from libraries; implies `–Mipa=inline`. The default is nolibinline.

[no]libopt
: allow recompiling and optimization of routines from libraries using IPA information. The default is nolibopt.

[no]localarg
: equivalent to arg plus externalization of local pointer targets. The default is nolocalarg.

main:<func>
: specify a function to appear as a global entry point; may appear multiple times; disables linking.

rsqrt
: Perform reciprocal square root (1/sqrt) using relaxed precision.

[no]pfo
: enable profile feedback information. The nopfo option is valid only immediately following the inline suboption. `-Mipa=inline,nopfo` tells IPA to ignore PFO information when deciding what functions to inline, if PFO information is available.

[no]ptr
: enable pointer disambiguation across procedure calls. The default is noptr.

[no]pure
: pure function detection. The default is nopure.

required
: return an error condition if IPA is inhibited for any reason, rather than the default behavior of linking without IPA optimization.

[no]reshape
: enables or disables Fortran inline with mismatched array shapes. Valid only immediately following the inline suboption.

safe:[<function>|<library>]
: declares that the named function, or all functions in the named library, are safe; a safe procedure does not call back into the known procedures and does not change any known global variables.

  Without `-Mipa=safe`, any unknown procedures will cause IPA to fail.

[no]safeall
: declares that all unknown procedures are safe; see `-Mipa=safe`. The default is nosafeall.

[no]shape
: perform Fortran 90 array shape propagation. The default is noshape.

summary
: only collect IPA summary information when compiling; this prevents IPA optimization of this file, but allows optimization for other files linked with this file.

[no]vestigial
: remove uncalled (vestigial) functions. The default is novestigial.

`-M[no]loop32`
: Aligns or does not align innermost loops on 32 byte boundaries with `-tp` barcelona.

  Small loops on barcelona may run fast if aligned on 32-byte boundaries; however, in practice, most assemblers do not yet implement efficient padding causing some programs to run more slowly with this default. Use `-Mloop32` on systems with an assembler tuned for barcleona. The default is `-Mnoloop32`.

–Mlre[=array | assoc | noassoc]
Enables loop-carried redundancy elimination, an optimization that can reduce the number of arithmetic operations and memory references in loops.

array
treat individual array element references as candidates for possible loop-carried redundancy elimination. The default is to eliminate only redundant expressions involving two or more operands.

assoc
allow expression re-association; specifying this sub-option can increase opportunities for loop-carried redundancy elimination but may alter numerical results.

noassoc
disallow expression re-association.

–Mnolre
Disables loop-carried redundancy elimination.

–Mnoframe
Eliminates operations that set up a true stack frame pointer for every function. With this option enabled, you cannot perform a traceback on the generated code and you cannot access local variables.

–Mnoi4
(pgf77 and pgf95 only) the compiler treats INTEGER variables as INTEGER*2.

–Mpfi[=indirect]
generate profile-feedback instrumentation; this includes extra code to collect run-time statistics and dump them to a trace file for use in a subsequent compilation.

When you use the indirect option,–Mpfi=indirect saves indirect function call targets.

–Mpfi must also appear when the program is linked. When the resulting program is executed, a profile feedback trace file pgfi.out is generated in the current working directory; see –Mpfo.

Note

Compiling and linking with –Mpfi adds significant runtime overhead to almost any executable. You should use executables compiled with –Mpfi only for execution of training runs.

–Mpfo[=indirect | nolayout]
enable profile-feedback optimizations; requires the presence of a pgfi.out profile-feedback trace file in the current working directory. See –Mpfi.

indirect
enable indirect function call inlining

nolayout
disable dynamic code layout.

–Mpre
enables partial redundancy elimination.

–Mprefetch[=option [,option...]]
enables generation of prefetch instructions on processors where they are supported. Possible values for option include:

d:m
set the fetch-ahead distance for prefetch instructions to m cache lines.

n:p
set the maximum number of prefetch instructions to generate for a given loop to p.

nta
use the prefetch instruction.

plain
use the prefetch instruction (default).

t0
use the prefetcht0 instruction.

w
use the AMD-specific prefetchw instruction.

–Mnoprefetch
Disables generation of prefetch instructions.

–M[no]propcond
Enables or disables constant propagation from assertions derived from equality conditionals.

The default is enabled.

–Mr8
(pgf77, pgf95 and pghpf only) the compiler promotes REAL variables and constants to DOUBLE PRECISION variables and constants, respectively. DOUBLE PRECISION elements are 8 bytes in length.

–Mnor8
(pgf77, pgf95 and pghpf only) the compiler does not promote REAL variables and constants to DOUBLE PRECISION. REAL variables will be single precision (4 bytes in length).

–Mr8intrinsics
(pgf77, and pgf95 only) the compiler treats the intrinsics CMPLX and REAL as DCMPLX and DBLE, respectively.

–Mnor8intrinsics
(pgf77, and pgf95 only) the compiler does not promote the intrinsics CMPLX and REAL to DCMPLX and DBLE, respectively.

–Msafeptr[=option[,option,...]]
(pgcc and pgcpp only) instructs the C/C++ compiler to override data dependencies between pointers of a given storage class. Possible values of option include:

all
assume all pointers and arrays are independent and safe for aggressive optimizations, and in particular that no pointers or arrays overlap or conflict with each other.

arg
: instructs the compiler that arrays and pointers are treated with the same copyin and copyout semantics as Fortran dummy arguments.

global
: instructs the compiler that global or external pointers and arrays do not overlap or conflict with each other and are independent.

local/auto
: instructs the compiler that local pointers and arrays do not overlap or conflict with each other and are independent.

static
: instructs the compiler that static pointers and arrays do not overlap or conflict with each other and are independent.

–Mscalarsse
: Use SSE/SSE2 instructions to perform scalar floating-point arithmetic. (This option is valid only on option `-tp` {p7 | k8-32 | k8-64} targets).

–Mnoscalarsse
: Do not use SSE/SSE2 instructions to perform scalar floating-point arithmetic; use x87 instructions instead. (This option is not valid in combination with the `-tp k8-64` option).

–Msmart
: instructs the compiler driver to invoke a post-pass assembly optimization utility.

–Mnosmart
: instructs the compiler not to invoke an AMD64-specific post-pass assembly optimization utility.

–M[no]traceback
: Adds debug information for runtime traceback for use with the environment variable `PGI_TERM`. By default, traceback is enabled for FORTRAN 77 and Fortran 90/95 and disabled for C and C++.

  Setting `setTRACEBACK=OFF;` in `siterc` or `.mypg*rc` also disables default traceback.

  Using ON instead of OFF enables default traceback.

–Munroll[=option [,option...]]
: invokes the loop unroller to execute multiple instances of the loop during each iteration. This also sets the optimization level to 2 if the level is set to less than 2, or if no `-O` or `-g` options are supplied. The option is one of the following:

  c:m
  : instructs the compiler to completely unroll loops with a constant loop count less than or equal to m, a supplied constant. If this value is not supplied, the m count is set to 4.

  m:<n>
  : instructs the compiler to unroll multi-block loops n times. This option is useful for loops that have conditional statements. If n is not supplied, then the default value is 4. The default setting is not to enable `–Munroll=m`.

n:<n>
: instructs the compiler to unroll single-block loops n times, a loop that is not completely unrolled, or has a non-constant loop count. If n is not supplied, the unroller computes the number of times a candidate loop is unrolled.

–Mnounroll
: instructs the compiler not to unroll loops.

-M[no]vect[=option [,option,...]]
: (disable) enable the code vectorizer, where option is one of the following:

  altcode
  : Instructs the vectorizer to generate alternate code (altcode) for vectorized loops when appropriate. For each vectorized loop the compiler decides whether to generate altcode and what type or types to generate, which may be any or all of: altcode without iteration peeling, altcode with non-temporal stores and other data cache optimizations, and altcode based on array alignments calculated dynamically at runtime. The compiler also determines suitable loop count and array alignment conditions for executing the altcode. This option is enabled by default.

  noaltcode
  : This disables alternate code generation for vectorized loops.

  assoc
  : Instructs the vectorizer to enable certain associativity conversions that can change the results of a computation due to roundoff error. A typical optimization is to change an arithmetic operation to an arithmetic operation that is mathematically correct, but can be computationally different, due to round-off error

  noassoc
  : Instructs the vectorizer to disable associativity conversions.

  cachesize:n
  : Instructs the vectorizer, when performing cache tiling optimizations, to assume a cache size of n. The default is set per processor type, either using the –tp switch or auto-detected from the host computer.

  [no]gather
  : Vectorize loops containing indirect array references, such as this one:

```
sum = 0.d0
do k=d(j),d(j+1)-1
sum = sum + a(k)*b(c(k))
enddo
```

    The default is gather.

  partial
  : Instructs the vectorizer to enable partial loop vectorization through innemost loop distribution.

  prefetch
  : Instructs the vectorizer to search for vectorizable loops and, wherever possible, make use of prefetch instructions.

[no]short
: Enable [disable] short vector operations. `-Mvect=short` enables generation of packed SSE instructions for short vector operations that arise from scalar code outside of loops or within the body of a loop iteration.

[no]sizelimit
: Generate vector code for all loops where possible regardless of the number of statements in the loop. This overrides a heuristic in the vectorizer that ordinarily prevents vectorization of loops with a number of statements that exceeds a certain threshold. The default is nosizelimit.

smallvect[:n]
: Instructs the vectorizer to assume that the maximum vector length is less than or equal to n. The vectorizer uses this information to eliminate generation of the stripmine loop for vectorized loops wherever possible. If the size n is omitted, the default is 100.

Note

No space is allowed on either side of the colon (:).

[no]sse
: Instructs the vectorizer to search for vectorizable loops and, wherever possible, make use of SSE, SSE2, and prefetch instructions. The default is nosse.

[no]uniform
: Instructs the vectorizer to perform the same optimizations in the vectorized and residual loops.

Note

This option may affect the performance of the residual loop.

–Mnovect
: instructs the compiler not to perform vectorization; can be used to override a previous instance of –Mvect on the command-line, in particular for cases in which –Mvect is included in an aggregate option such as –fastsse.

–Mvect=[*option*]
: instructs the compiler to enable loop vectorization, where option is one of the following:

partial
: Instructs the vectorizer to enable partial loop vectorization through innemost loop distribution.

[no]short
: Enable [disable] short vector operations. Enables [disables] generation of packed SSE instructions for short vector operations that arise from scalar code outside of loops or within the body of a loop iteration.

–Mnovintr
: instructs the compiler not to perform idiom recognition or introduce calls to hand-optimized vector functions.

Miscellaneous Controls

Default: For arguments that you do not specify, the default miscellaneous options are as follows:

inform nobounds nolist warn

Usage: In the following example, the compiler includes Fortran source code with the assembly code.

```
$ pgf95 -Manno -S myprog.f
```

In the following example, the assembler does not delete the assembly file `myprog.s` after the assembly pass.

```
$ pgf95 -Mkeepasm myprog.f
```

In the following example, the compiler displays information about inlined functions with fewer than approximately 20 source lines in the source file `myprog.f`.

```
$ pgf95 -Minfo=inline -Minline=20 myprog.f
```

In the following example, the compiler creates the listing file `myprog.lst`.

```
$ pgf95 -Mlist myprog.f
```

In the following example, array bounds checking is enabled.

```
$ pgf95 -Mbounds myprog.f
```

Related options: –m, –S, –V, –v

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Manno
: annotate the generated assembly code with source code. Implies `–Mkeepasm` .

–Mbounds
: enables array bounds checking. If an array is an assumed size array, the bounds checking only applies to the lower bound. If an array bounds violation occurs during execution, an error message describing the error is printed and the program terminates. The text of the error message includes the name of the array, the location where the error occurred (the source file and the line number in the source), and information about the out of bounds subscript (its value, its lower and upper bounds, and its dimension). The following is a sample error message:

```
PGFTN-F-Subscript out of range for array a (a.f: 2)
subscript=3, lower bound=1, upper bound=2, dimension=2
```

–Mnobounds
: disables array bounds checking.

–Mbyteswapio
: swap byte-order from big-endian to little-endian or vice versa upon input/output of Fortran unformatted data files.

–Mchkfpstk (32-bit only)
: instructs the compiler to check for internal consistency of the x87 floating-point stack in the prologue of a function and after returning from a function or subroutine call. Floating-point stack corruption may occur in many ways, one of which is Fortran code calling floating-point functions as subroutines (i.e., with the CALL statement). If the `PGI_CONTINUE` environment variable is set upon execution of a program

compiled with `-Mchkfpstk`, the stack will be automatically cleaned up and execution will continue. There is a performance penalty associated with the stack cleanup. If `PGI_CONTINUE` is set to verbose, the stack will be automatically cleaned up and execution will continue after printing the warning message.

Note

This switch is only valid for 32-bit. On 64-bit it is ignored.

–Mchkptr
: instructs the compiler to check for pointers that are dereferenced while initialized to NULL (pgf95 and pghpf only).

–Mchkstk
: instructs the compiler to check the stack for available space in the prologue of a function and before the start of a parallel region. Prints a warning message and aborts the program gracefully if stack space is insufficient. Useful when many local and private variables are declared in an OpenMP program.

  If the user also sets the `PGI_STACK_USAGE` environment variable to any value, then the program displays the stack space allocated and used after the program exits. For example, you might see something similar to the following message:

```
thread 0 stack: max 8180KB, used 48KB
```

  This message indicates that the program used 48KB of a 8180KB allocated stack. For more information on the `PGI_STACK_USAGE`, refer to "PGI_STACK_USAGE," on page 135.

  This information is useful when you want to explicitly set a reserved and committed stack size for your programs, such as using the `-stack` option on Windows.

–Mcpp[=option [,option,...]]
: run the PGI cpp-like preprocessor without execution of any subsequent compilation steps. This option is useful for generating dependence information to be included in makefiles.

Note

Only one of the m, md, mm or mmd options can be present; if multiple of these options are listed, the last one listed is accepted and the others are ignored.

The option is one or more of the following:

m
: print makefile dependencies to stdout.

md
: print makefile dependencies to `filename.d`, where filename is the root name of the input file being processed.

mm
: print makefile dependencies to stdout, ignoring system include files.

mmd
: print makefile dependencies to `filename.d`, where filename is the root name of the input file being processed, ignoring system include files.

[no]comment
(don't) retain comments in output.

[suffix:]<suff>
use <suff> as the suffix of the output file containing makefile dependencies.

–Mdll
This Windows-only flag has been deprecated. Refer to `-Bdynamic`. This flag was used to link with the DLL versions of the runtime libraries, and it was required when linking with any DLL built by any of The Portland Group compilers. This option implied –D_DLL, which defines the preprocessor symbol _DLL.

–Mgccbug[s]
match the behavior of certain gcc bugs.

–Miface[=option]
adjusts the calling conventions for Fortran, where option is one of the following:

unix
(Win32 only) uses UNIX calling conventions, no trailing underscores.

cref
uses CREF calling conventions, no trailing underscores.

mixed_str_len_arg
places the lengths of character arguments immediately after their corresponding argument. Has affect only with the CREF calling convention.

nomixed_str_len_arg
places the lengths of character arguments at the end of the argument list. Has affect only with the CREF calling convention.

–Minfo[=option [,option,...]]
instructs the compiler to produce information on standard error, where option is one of the following:

all
instructs the compiler to produce all available `-Minfo` information. Implies a number of suboptions:

```
-Mneginfo=accel,inline,ipa,loop,lre,mp,opt,par,vect
```

accel
instructs the compiler to enable accelerator information.

ccff
instructs the compiler to append common compiler feedback format information, such as optimization information, to the object file.

ftn
instructs the compiler to enable Fortran-specific information.

hpf
instructs the compiler to enable HPF-specific information.

inline
instructs the compiler to display information about extracted or inlined functions. This option is not useful without either the `-Mextract` or `-Minline` option.

intensity
: instructs the compiler to provide informational messages about the intensity of the loop. Specify <n> to get messages on nested loops.

  - For floating point loops, intensity is defined as the number of floating point operations divided by the number of floating point loads and stores.
  - For integer loops, the loop intensity is defined as the total number of integer arithmetic operations, which may include updates of loop counts and addresses, divided by the total number of integer loads and stores.
  - By default, the messages just apply to innermost loops.

ipa
: instructs the compiler to display information about interprocedural optimizations.

loop
: instructs the compiler to display information about loops, such as information on vectorization.

lre
: instructs the compiler to enable LRE, loop-carried redundancy elimination, information.

mp
: instructs the compiler to display information about parallelization.

opt
: instructs the compiler to display information about optimization.

par
: instructs the compiler to enable parallelizer information.

pfo
: instructs the compiler to enable profile feedback information.

time
: instructs the compiler to display compilation statistics.

unroll
: instructs the compiler to display information about loop unrolling.

vect
: instructs the compiler to enable vectorizer information.

`–Minform=level`
: instructs the compiler to display error messages at the specified and higher levels, where level is one of the following:

  fatal
  : instructs the compiler to display fatal error messages.

  [no]file
  : instructs the compiler to print or not print source file names as they are compiled. The default is to print the names: `–Minform=file`.

inform
: instructs the compiler to display all error messages (inform, warn, severe and fatal).

severe
: instructs the compiler to display severe and fatal error messages.

warn
: instructs the compiler to display warning, severe and fatal error messages.

`–Minstrumentation=option`
: specifies the level of instrumentation calls generated. This option implies `-Minfo=ccff`, `-Mframe`.

`option` is one of the following:

level
: specifies the level of instrumentation calls generated.

function (default)
: generates instrumentation calls for entry and exit to functions.

  Just after function entry and just before function exit, the following profiling functions are called with the address of the current function and its call site. (linux86-64 only).

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit (void *this_fn, void *call_site);
```

  In these calls, the first argument is the address of the start of the current function.

`–Mkeepasm`
: instructs the compiler to keep the assembly file as compilation continues. Normally, the assembler deletes this file when it is finished. The assembly file has the same filename as the source file, but with a .s extension.

`–Mlist`
: instructs the compiler to create a listing file. The listing file is `filename.lst`, where the name of the source file is `filename.f`.

`–Mmakedll`
: (Windows only) generate a dynamic link library (DLL).

`–Mmakeimplib`
: (Windows only) generate an import library for a DLL without creating the DLL. When used without -def:deffile, passes the -def switch to the librarian without a deffile.

`–Mnames=lowercase|uppercase`
: specifies the case for the names of Fortran externals .

- lowercase - Use lowercase for Fortran externals.
- uppercase - Use uppercase for Fortran externals.

`–Mneginfo[=option [,option,...]]`
: instructs the compiler to produce information on standard error, where option is one of the following:

all
: instructs the compiler to produce all available information on why various optimizations are not performed.

accel
: instructs the compiler to enable accelerator information.

ccff
: instructs the compiler to append information, such as optimization information, to the object file.

concur
: instructs the compiler to produce all available information on why loops are not automatically parallelized. In particular, if a loop is not parallelized due to potential data dependence, the variable(s) that cause the potential dependence are listed in the messages that you see when using the option `–Mneginfo`.

ftn
: instructs the compiler to enable Fortran-specific information.

hpf
: instructs the compiler to enable HPF-specific information.

inline
: instructs the compiler to display information about extracted or inlined functions. This option is not useful without either the `–Mextract` or `–Minline` option.

ipa
: instructs the compiler to display information about interprocedural optimizations.

loop
: instructs the compiler to display information about loops, such as information on vectorization.

lre
: instructs the compiler to enable LRE, loop-carried redundancy elimination, information.

mp
: instructs the compiler to display information about parallelization.

opt
: instructs the compiler to display information about optimization.

par
: instructs the compiler to enable parallelizer information.

pfo
: instructs the compiler to enable profile feedback information.

vect
: instructs the compiler to enable vectorizer information.

`–Mnolist`
: the compiler does not create a listing file. This is the default.

–Mnoopenmp
: when used in combination with the –mp option, the compiler ignores OpenMP parallelization directives or pragmas, but still processes SGI-style parallelization directives or pragmas.

–Mnosgimp
: when used in combination with the –mp option, the compiler ignores SGI-style parallelization directives or pragmas, but still processes OpenMP parallelization directives or pragmas.

–Mnopgdllmain
: (Windows only) do not link the module containing the default DllMain() into the DLL. This flag applies to building DLLs with the PGF95 and PGHPF compilers. If you want to replace the default DllMain() routine with a custom DllMain(), use this flag and add the object containing the custom DllMain() to the link line. The latest version of the default DllMain() used by PGF95 and PGHPF is included in the Release Notes for each release; the PGF95- and PGHPF-specific code in this routine must be incorporated into the custom version of DllMain() to ensure the appropriate function of your DLL.

–Mnorpath
: (Linux only) Do not add –rpath to the link line.

–Mpreprocess
: perform cpp-like preprocessing on assembly and Fortran input source files.

–Mwritable_strings
: stores string constants in the writable data segment.

> **Note**
>
> Options –Xs and –Xst include –Mwritable_strings.

Chapter 17. OpenMP Reference Information

The PGF77 and PGF95 Fortran compilers support the OpenMP Fortran Application Program Interface. The PGCC ANSI C and C++ compilers support the OpenMP C/C++ Application Program Interface.

This chapter contains detailed descriptions of each of the OpenMP Fortran directives and C/C++ pragmas that PGI supports. In addition, the section "Directive and Pragma Clauses," on page 303 contains information about the clauses associated with these directives and pragmas.

Tasks

Every part of an OpenMP program is part of a task. "Task Overview," on page 58 provides a general overview of tasks and general terminology associated with tasks. This section provides more detailed information about tasks, including tasks scheduling points and the task construct.

Task Characteristics and Activities

A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

Task Scheduling Points

PGI currently supports four task scheduling points: at the beginning of a task, at the end of a task, a taskwait, and at a barrier.

- Beginning of a task.

  At the beginning of a task, the task can be executed immediately or registered for later execution. A programmer-specified "if" clause that is FALSE forces immediate execution of the task. The implementation can also force immediate execution; for example, a task within a task is never registered for later execution, it executes immediately.

- End of a task

  At the end of a task, the behavior of the scheduling point depends on how the task was executed. If the task was immediately executed, execution continues to the next statement. If it was previously registered and is being executed "out of sequence", control returns to where the task was executed.

- Taskwait

  A taskwait executes all registered tasks at the time it is called. In addition to executing all tasks registered by the calling thread, it also executes tasks previously registered by other threads. Let's take a quick look at this process; suppose the following is true:

  - Thread 0 called taskwait and is executing tasks.
  - Thread 1 is registering tasks.

  Depending on the timing between thread 0 and thread 1, thread 0 may execute none of the tasks, all of the tasks, or some of tasks.

  > **Note**
  >
  > Taskwait waits only for immediate children tasks, not for descendant tasks. You can achieve waiting on descendants but ensuring that each child also waits on its children.

- Barrier

  A barrier can be explicit or implicit. An example of an implicit barrier is the end of a parallel region.

  The barrier effectively contains taskwaits. All threads must arrive at the barrier for the barrier to complete. This rule guarantees that all tasks have been executed at the completion of the barrier.

Task Construct

A task construct is a task directive plus a structured block, with the following syntax:

```
#pragma omp task [clause[[,]clause] ...]
     structured-block
```

where clause can be one of the following:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
```

Consider the following simple example of a program using tasks. This example illustrates the difference between registering tasks and executing tasks, a concept that is fundamental to understanding tasks.

This program contains a parallel region that contains a single region. The single region contains a loop that registers 10 tasks. Before reading the explanation that follows the example, consider what happens if you use four threads with this example.

Example 17.1. OpenMP Task C Example

```
int
main(int argc,char *argv[])
{
     int i;
#pragma omp parallel private(i)
{
#pragma omp single
{
     for(i=0;i<10;i++) {
         sleep(i%2);
         printf("task %2d registered by thread %d\n",i,
                 omp_get_thread_num());
#pragma omp task firstprivate(i)
{
         sleep(i%5);
         printf("task %2d executed by thread %d\n",i,
                 omp_get_thread_num());
} /* end task */
     } /* end for */
} /* end single */
} /* end parallel */
} /* end main */
```

Example 17.2. OpenMP Task Fortran Example

```
     PROGRAM MAIN
     INTEGER I
     INTEGER omp_get_thread_num
!$OMP PARALLEL PRIVATE(I)
!$OMP SINGLE
     DO I = 1, 10
        CALL SLEEP(MOD(I,2))
        PRINT *,"TASK ",I," REGISTERED BY THREAD ",omp_get_thread_num()
!$OMP TASK FIRSTPRIVATE(I)
        CALL SLEEP(MOD(I,5))
        PRINT *,"TASK ",I," EXECUTED BY THREAD   ",omp_get_thread_num()
!$OMP END TASK
     ENDDO
!$OMP END SINGLE
!$OMP END PARALLEL
     END
```

If you run this program with four threads, 0 through 3, one thread is in the single region registering tasks. The other three threads are in the implied barrier at the end of the single region executing tasks. Further, when the thread executing the single region completes registering the tasks, it joins the other threads and executes tasks.

The program includes calls to `sleep` to slow the program and allow all threads to participate.

The output for the Fortran example is similar to the following. In this output, thread 1 was registering tasks while the other three threads - 0,2, and 3 - were executing tasks When all 10 tasks were registered, thread 1 began executing tasks as well.

```
TASK 1  REGISTERED BY THREAD 1
TASK 2  REGISTERED BY THREAD 1
TASK 1  EXECUTED BY THREAD 0
TASK 3  REGISTERED BY THREAD 1
TASK 4  REGISTERED BY THREAD 1
TASK 2  EXECUTED BY THREAD 3
TASK 5  REGISTERED BY THREAD 1
TASK 6  REGISTERED BY THREAD 1
TASK 6  EXECUTED BY THREAD 3
TASK 5  EXECUTED BY THREAD 3
TASK 7  REGISTERED BY THREAD 1
TASK 8  REGISTERED BY THREAD 1
TASK 3  EXECUTED BY THREAD 0
TASK 9  REGISTERED BY THREAD 1
TASK 10  REGISTERED BY THREAD 1
TASK 10  EXECUTED BY THREAD 1
TASK 4  EXECUTED BY THREAD 2
TASK 7  EXECUTED BY THREAD 0
TASK 8  EXECUTED BY THREAD 3
TASK 9  EXECUTED BY THREAD 1
```

Parallelization Directives and Pragmas

Parallelization directives, as described in Chapter 5, "*Using OpenMP*", are comments in a program that are interpreted by the PGI Fortran compilers when the option `-mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

Parallelization pragmas are #pragma statements in a C or C++ program that are interpreted by the PGCC C and C++ compilers when the option -mp is specified on the command line. The form of a parallelization pragma is:

```
#pragma omp pragma_name [clauses]
```

The examples given with each section use the routines `omp_get_num_threads()` and `omp_get_thread_num()`. They return the number of threads currently in the team executing the parallel region and the thread number within the team, respectively. For more information, refer to "Run-time Library Routines," on page 65.

Note

Directives which are presented in pairs must be used in pairs.

This section describes the details of these directives and pragmas that were summarized in Chapter 5, "*Using OpenMP*". For each directive and pragma, this section describes the overall purpose, the syntax, the clauses associated with it, the usage, and examples of how to use it.

ATOMIC and atomic

The OpenMP ATOMIC directive or the omp critical pragma is semantically equivalent to a single statement in a CRITICAL...END CRITICAL directive or the omp critical pragma.

Syntax:

`!$OMP ATOMIC`	`#pragma omp atomic` `< C/C++ expression statement >`

Usage:

The ATOMIC directive is semantically equivalent to enclosing the following single statement in a CRITICAL / END CRITICAL directive pair. The omp atomic pragma is semantically equivalent to subjecting the following single C/C++ expression statement to an omp critical pragma.

The statements must be one of the following forms:

For Directives:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

For Pragmas:

```
x <binary_operator>= expr
x++
++x
x--
--x
```

where `x` is a scalar variable of intrinsic type, `expr` is a scalar expression that does not reference `x`, intrinsic is one of MAX, MIN, IAND, IOR, or IEOR, operator is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV., and <binary_operator> is not overloaded and is one of +, *, -, /, &, ^, |, << or >>.

BARRIER and barrier

The OpenMP BARRIER directive defines a point in a program where each thread waits for all other threads to arrive before continuing with program execution.

Syntax:

`!$OMP BARRIER`	`#pragma omp barrier`

Usage:

There may be occasions in a parallel region when it is necessary that all threads complete work to that point before any thread is allowed to continue. The BARRIER directive or omp barrier pragma synchronizes all threads at such a point in a program. Multiple barrier points are allowed within a parallel region. The BARRIER directive and omp barrier pragma must either be executed by all threads executing the parallel region or by none of them.

CRITICAL ... END CRITICAL and critical

The CRITICAL...END CRITICAL directive and omp critical pragma require a thread to wait until no other thread is executing within a critical section.

Syntax:

`!$OMP CRITICAL [(name)]`
` < Fortran code executed in body`
`of critical section >`
`!$OMP END CRITICAL [(name)]`
	`#pragma omp critical [(name)]`
	`< C/C++ structured block >`

Usage:

Within a parallel region, there may exist subregions of code that will not execute properly when executed by multiple threads simultaneously. This issue is often due to a shared variable that is written and then read again.

The CRITICAL... END CRITICAL directive pair and the omp critical pragma define a subsection of code within a parallel region, referred to as a critical section, which is executed one thread at a time.

The first thread to arrive at a critical section is the first to execute the code within the section. The second thread to arrive does not begin execution of statements in the critical section until the first thread exits the critical section. Likewise, each of the remaining threads wait its turn to execute the statements in the critical section.

You can use the optional name argument to identify the critical region. Names that identify critical regions have external linkage and are in a name space separate from the name spaces used by labels, tags, members, and ordinary identifiers. If a name argument appears on a CRITICAL directive, the same name must appear on the END CRITICAL directive.

Note

Critical sections cannot be nested, and any such specifications are ignored. Branching into or out of a critical section is illegal.

Fortran Example of Critical...End Critical directive:

```
 PROGRAM CRITICAL_USE
 REAL A(100,100),MX, LMX
 INTEGER I, J
 MX = -1.0
 LMX = -1.0
 CALL RANDOM_SEED()
 CALL RANDOM_NUMBER(A)
!$OMP PARALLEL PRIVATE(I), FIRSTPRIVATE(LMX)
!$OMP DO
 DO J=1,100
 DO I=1,100
 LMX = MAX(A(I,J),LMX)
 ENDDO
 ENDDO
!$OMP CRITICAL
 MX = MAX(MX,LMX)
!$OMP END CRITICAL
!$OMP END PARALLEL
 PRINT *,"MAX VALUE OF A IS ", MX
 END
```

C Example of **omp critical** pragma

```
#include <stdlib.h>
main(){
int a[100][100], mx=-1,lmx=-1, i, j;
```

```
 for (j=0; j<100; j++)
 for (i=0; i<100; i++)
 a[i][j]=1+(int)(10.0*rand()/(RAND_MAX+1.0));
#pragma omp parallel private(i) firstprivate(lmx)
 {
#pragma omp for
 for (j=0; j<100; j++)
 for (i=0; i<100; i++)
 lmx = (lmx > a[i][j]) ? lmx : a[i][j];
#pragma omp critical
 mx = (mx > lmx) ? mx : lmx;
 }
 printf ("max value of a is %d\n",mx);
}
```

This program could also be implemented without the critical region by declaring `MX` as a reduction variable and performing the `MAX` calculation in the loop using `MX` directly rather than using `LMX`. Refer to "PARALLEL ... END PARALLEL and parallel " and "DO...END DO and for " for more information on how to use the REDUCTION clause on a parallel DO loop.

C$DOACROSS

The C$DOACROSS directive, while not part of the OpenMP standard, is supported for compatibility with programs parallelized using legacy SGI-style directives.

Syntax:

```
C$DOACROSS [ Clauses ]
< Fortran DO loop to be executed
 in parallel >
```

```
#pragma omp parallel [clauses]
< C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
[{PRIVATE | LOCAL} (list)]	private | local(list)
[{SHARED | SHARE} (list)]	shared | share(list)
[MP_SCHEDTYPE={SIMPLE | INTERLEAVE}]	mp_schedule (simple | interleave)
[CHUNK=<integer_expression>]	chunk=<integer_expression>
[IF (logical_expression)]	if (logical_expression)

Usage:

The C$DOACROSS directive has the effect of a combined parallel region and parallel DO loop applied to the loop immediately following the directive. It is very similar to the OpenMP PARALLEL DO directive, but provides for backward compatibility with codes parallelized for SGI systems prior to the OpenMP standardization effort.

The C$DOACROSS directive must not appear within a parallel region. It is a shorthand notation that tells the compiler to parallelize the loop to which it applies, even though that loop is not contained within a parallel region. While this syntax is more convenient, it should be noted that if multiple successive DO loops are to be parallelized it is more efficient to define a single enclosing parallel region and parallelize each loop using the OpenMP DO directive.

A variable declared PRIVATE or LOCAL to a C$DOACROSS loop is treated the same as a private variable in a parallel region or DO. A variable declared SHARED or SHARE to a C$DOACROSS loop is shared among the threads, meaning that only 1 copy of the variable exists to be used and/or modified by all of the threads. This is equivalent to the default status of a variable that is not listed as PRIVATE in a parallel region or DO. This same default status is used in C$DOACROSS loops as well. For more information on clauses, refer to "Directive and Pragma Clauses," on page 303.

DO...END DO and for

The OpenMP DO...END DO directive and omp for pragma support parallel execution and the distribution of loop iterations across available threads in a parallel region.

Syntax:

```
!$OMP DO [Clauses]
< Fortran DO loop to be executed
 in parallel
 !$OMP END DO [NOWAIT]
```

```
#pragma omp for [Clauses]
< C/C++ for loop to be executed
in parallel >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	lastprivate(list)
REDUCTION({operator | intrinsic } : list)	reduction(operator: list)
SCHEDULE (type [, chunk])	schedule (kind[, chunk])
COLLAPSE (n)	collapse (n)
ORDERED	ordered
	nowait

Usage:

The real purpose of supporting parallel execution is the distribution of work across the available threads. The DO... END DO directive pair and the omp for pragma provide a convenient mechanism for the distribution of loop iterations across the available threads in a parallel region.

While you can explicitly manage work distribution with constructs such as the following one, these constructs are not in the form of directives or pragmas.

Examples:

For Directives:

```
IF (omp_get_thread_num() .EQ. 0)
THEN
   ...
ELSE IF (omp_get_thread_num() .EQ. 1)
THEN
   ...
ENDIF
```

For Pragmas:

```
if (omp_get_thread_num() == 0) {
 ...
}
else if (omp_get_thread_num() == 1) {
 ...
}
```

Tips

Remember these items about clauses in the DO...END DO directives and omp for pragmas:

- Variables declared in a PRIVATE list are treated as private to each thread participating in parallel execution of the loop, meaning that a separate copy of the variable exists with each thread.
- Variables declared in a FIRSTPRIVATE list are PRIVATE, and are initialized from the original object existing before the construct.
- Variables declared in a LASTPRIVATE list are PRIVATE, and the thread that executes the sequentially last iteration updates the version of the object that existed before the construct.
- The REDUCTION clause for the directive is described in "REDUCTION," on page 306 and the reduction clause for the pragma is described in "Directive and Pragma Clauses," on page 303.
- The SCHEDULE clause specifies how iterations of the DO or for loop are divided up between threads. For more information on this clause, refer to "SCHEDULE," on page 307.
- If ORDERED code blocks are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. For more information on ORDERED code blocks, refer to "ORDERED and ordered ".
- The DO... END DO directive pair directs the compiler to distribute the iterative DO loop immediately following the !$OMP DO directive across the threads available to the program. The DO loop is executed in parallel by the team that was started by an enclosing parallel region. If the !$OMP END DO directive is not specified, the !$OMP DO is assumed to end with the enclosed DO loop. DO... END DO directive pairs may not be nested. Branching into or out of a !$OMP DO loop is not supported.
- The **omp for** pragma directs the compiler to distribute the iterative for loop immediately following across the threads available to the program. The for loop is executed in parallel by the team that was started by an enclosing parallel region. Branching into or out of an omp for loop is not supported, and omp for pragmas may not be nested.
- By default, there is an implicit barrier after the end of the parallel loop; the first thread to complete its portion of the work waits until the other threads have finished their portion of work. If NOWAIT is specified, the threads will not synchronize at the end of the parallel loop.

In addition to the preceding items, remember these items about !$OMP DO loops and **omp for** loops:

- The DO loop index variable is always private.
- The for loop index variable is always private.
- !$OMP DO loops and omp for loops must be executed by all threads participating in the parallel region or none at all.
- The END DO directive is optional, but if it is present it must appear immediately after the end of the enclosed DO loop.
- The for loop must be a structured block and its execution must not be terminated by break.
- Values of the loop control expressions and the chunk expressions must be the same for all threads executing the loop.

Examples:

Fortran Example of **DO...END DO** directive

```
PROGRAM DO_USE
   REAL A(1000), B(1000)
   DO I=1,1000
      B(I) = FLOAT(I)
   ENDDO
!$OMP PARALLEL
!$OMP DO
   DO I=1,1000
      A(I) = SQRT(B(I));
   ENDDO
  ...
!$OMP END PARALLEL
   ...
   END
```

C Example of **omp for** pragma

```
#include <stdio.h>
#include <math.h>
main(){
float a[1000], b[1000];
int i;
 for (i=0; i<1000; i++)
 b[i] = i;
#pragma omp parallel
 {
#pragma omp for
 for (i=0; i<1000; i++)
 a[i] = sqrt(b[i]);
 ...
 }
 ...
}
```

FLUSH and flush

The OpenMP FLUSH directive and omp flush pragma ensure that processor-visible data item are written back to memory at the point at which the directive appears.

Syntax:

```
!$OMP FLUSH [(list)]
```

```
#pragma omp flush [(list)]
```

Usage:

The OpenMP FLUSH directive ensures that all processor-visible data items, or only those specified in `list`, when it is present, are written back to memory at the point at which the directive or pragma appears.

MASTER ... END MASTER and master

The MASTER...END MASTER directive and omp master pragma allow the user to designate code that must execute on a master thread and that is skipped by other threads in the team of threads.

Syntax:

```
!$OMP MASTER
< Fortran code executed in body of
 MASTER section >
!$OMP END MASTER
```

```
#pragma omp master
< C/C++ structured block >
```

Usage:

A master thread is a single thread of control that begins an OpenMP program and which is present for the duration of the program. In a parallel region of code, there may be a sub-region of code that should execute only on the master thread. Instead of ending the parallel region before this subregion and then starting it up

again after this subregion, the MASTER... END MASTER directive pair or omp master pragma allows the user to conveniently designate code that executes on the master thread and is skipped by the other threads.

- There is no implied barrier on entry to or exit from a master section of code.
- Nested master sections are ignored.
- Branching into or out of a master section is not supported.

Examples:

Example of Fortran **MASTER...END MASTER** directive

```
PROGRAM MASTER_USE
     INTEGER A(0:1)
     INTEGER omp_get_thread_num
     A=-1
!$OMP PARALLEL
     A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP MASTER
     PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END MASTER
!$OMP END PARALLEL
     PRINT *, "A(0)=", A(0), " A(1)=", A(1)
 END
```

Example of **omp master** pragma

```
#include <stdio.h>
#include <omp.h>
main(){
 int a[2]={-1,-1};
#pragma omp parallel
 {
 a[omp_get_thread_num()] = omp_get_thread_num();
#pragma omp master
 printf("YOU SHOULD ONLY SEE THIS ONCE\n");
 }
 printf("a[0]=%d, a[1]=%d\n",a[0],a[1]);
}
```

ORDERED and ordered

The OpenMP ORDERED directive and omp ordered pragma allow the user to identify a portion of code within an ordered code block that must be executed in the original, sequential order, while allowing parallel execution of statements outside the code block.

Syntax:

```
!$OMP ORDERED
< Fortran code block executed
 by processor >
!$OMP END ORDERED
```

```
#pragma omp ordered
< C/C++ structured block >
```

Usage:

The ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive that includes the ORDERED clause. The ordered pragma can appear only in the dynamic extent of a for or parallel for pragma that includes the ordered clause. The structured code block between the ORDERED / END ORDERED directives or after the ordered pragma is executed by only one thread at a time, and in the order of the loop iterations. This sequentializes the ordered code block while allowing parallel execution of statements outside the code block. The following additional restrictions apply to the ORDERED directive and ordered pragma:

- The ordered code block must be a structured block.
- It is illegal to branch into or out of the block.
- A given iteration of a loop with a DO directive or omp for pragma cannot execute the same ORDERED directive or omp ordered pragma more than once, and cannot execute more than one ORDERED directive or omp ordered pragma.

PARALLEL ... END PARALLEL and parallel

The OpenMP PARALLEL...END PARALLEL directive and OpenMP omp parallel pragma support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.

Syntax:

```
!$OMP PARALLEL [Clauses]
   < Fortran code executed in body
 of parallel region >
!$OMP END PARALLEL
```

```
#pragma omp parallel [clauses]
   < C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
SHARED(list)	shared(list)
DEFAULT(PRIVATE | SHARED | NONE)	default(shared | none)
FIRSTPRIVATE(list)	firstprivate(list)
REDUCTION([{operator | intrinsic}:] list)	reduction(operator: list)
COPYIN(list)	copyin (list)
IF(scalar_logical_expression)	if (scalar_expression)
NUM_THREADS(scalar_integer_expression)	num_threads(scalar_integer_expression)

Usage:

This directive pair or pragma declares a region of parallel execution. It directs the compiler to create an executable in which the statements within the structured block, such as between PARALLEL and PARALLEL END for directives, are executed by multiple lightweight threads. The code that lies within this structured block is called a *parallel region*.

The OpenMP parallelization directives or pragmas support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered. At the entrance to the parallel region, a system-dependent number of symmetric parallel threads begin executing all statements in the parallel region

redundantly. These threads share work by means of work-sharing constructs such as parallel DO loops or FOR loops.

- The number of threads in the team is controlled by the `OMP_NUM_THREADS` environment variable. If `OMP_NUM_THREADS` is not defined, the program executes parallel regions using only one processor.
- Branching into or out of a parallel region is not supported.
- All other shared-memory parallelization directives or pragmas must occur within the scope of a parallel region. Nested PARALLEL... END PARALLEL directive pairs or omp parallel pragmas are not supported and are ignored.
- There is an implicit barrier at the end of the parallel region, which, in the directive, is denoted by the END PARALLEL directive. When all threads have completed execution of the parallel region, a single thread resumes execution of the statements that follow.

Note

By default, there is no work distribution in a parallel region. Each active thread executes the entire region redundantly until it encounters a directive or pragma that specifies work distribution. For work distribution, see the DO, PARALLEL DO, or DOACROSS directives or the omp for pragma.

Examples:

PARALLEL...END PARALLEL directive example:

```
 PROGRAM WHICH_PROCESSOR_AM_I
   INTEGER A(0:1)
   INTEGER omp_get_thread_num
   A(0) = -1
   A(1) = -1
!$OMP PARALLEL
   A(omp_get_thread_num())
        = omp_get_thread_num()
!$OMP END PARALLEL
   PRINT *, "A(0)=",A(0),
            " A(1)=",A(1)
 END
```

omp parallel pragma example

```
#include <stdio.h>
#include <omp.h>
main(){
   int a[2]={-1,-1};
#pragma omp parallel
  {
     a[omp_get_thread_num()] =
 omp_get_thread_num();
  }
  printf("a[0] = %d,
a[1] = %d",a[0],a[1]);
}
```

Clause Usage:

COPYIN: The COPYIN clause applies only to THREADPRIVATE common blocks. In the presence of the COPYIN clause, data from the master thread's copy of the common block is copied to the THREADPRIVATE copies upon entry to the parallel region.

IF: In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to .TRUE.. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

NUM_THREADS: If the NUM_THREADS clause is present, the corresponding `scalar_integer_expression` must evaluate to a positive integer value. This value sets the maximum number of threads used during execution of the parallel region. A NUM_THREADS clause overrides either a previous call to the library routine omp_set_num_threads() or the setting of the OMP_NUM_THREADS environment variable.

PARALLEL DO

The OpenMP PARALLEL DO directive is a shortcut for a PARALLEL region that contains a single DO directive.

Note

The OpenMP PARALLEL DO or DO directive must be immediately followed by a DO statement (as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the PARALLEL DO or DO directive and the DO statement, the compiler issues a syntax error.

Syntax:

```
!$OMP PARALLEL DO [CLAUSES]
< Fortran DO loop to be executed
 in parallel >
[!$OMP END PARALLEL DO]
```

```
#pragma omp parallel [clauses]
< C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
SHARED(list)	shared(list)
DEFAULT(PRIVATE | SHARED | NONE)	default(shared | none)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	reduction(operator: list)
REDUCTION([{operator | intrinsic}:] list)	copyin (list)
COPYIN(list)	if (scalar_expression)
IF(scalar_logical_expression)	num_threads(scalar_integer_expression)
NUM_THREADS(scalar_integer_expression)	
SCHEDULE (type [, chunk])	
COLLAPSE (n)	
ORDERED	

Usage:

The semantics of the PARALLEL DO directive are identical to those of a parallel region containing only a single parallel DO loop and directive. The available clauses are the same as those defined in “PARALLEL ... END PARALLEL and parallel ,” on page 294 and “DO...END DO and for ”.

Note

The END PARALLEL DO directive is optional.

PARALLEL SECTIONS and parallel sections

The OpenMP PARALLEL SECTIONS / END SECTIONS directive pair and the omp parallel sections pragma define tasks to be executed in parallel; that is, they define a non-iterative work-sharing construct without the need to define an enclosing parallel region.

Syntax:

```
!$OMP PARALLEL SECTIONS [CLAUSES]
[!$OMP SECTION]
< Fortran code block executed
 by processor i >
[!$OMP SECTION]
< Fortran code block executed
 by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

```
#pragma omp parallel sections [clauses]
{
 [#pragma omp section]
 < C/C++ structured block executed
 by processor i >
 [#pragma omp section]
 < C/C++ structured block executed
 by processor j >
 ...
}
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
SHARED(list)	shared(list)
DEFAULT(PRIVATE | SHARED | NONE)	default(shared | none)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	lastprivate (list)
REDUCTION({operator | intrinsic} : list)	reduction({operator: list)
COPYIN (list)	copyin (list)
IF(scalar_logical_expression)	if (scalar_expression)
NUM_THREADS(scalar_integer_expression)	num_threads(scalar_integer_expression)
	nowait

Usage:

The PARALLEL SECTIONS / END SECTIONS directive pair and the omp parallel sections pragma define a non-iterative work-sharing construct without the need to define an enclosing parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors will execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing PARALLEL SECTIONS / END SECTIONS directives. In addition, the code within the PARALLEL SECTIONS / END SECTIONS directives must be a structured block, and the code in each SECTION must be a structured block.

Semantics are identical to a parallel region containing only an omp sections pragma and the associated structured block. The available clauses are as defined in “PARALLEL ... END PARALLEL and parallel ,” on page 294 and “DO...END DO and for ”.

PARALLEL WORKSHARE ... END PARALLEL WORKSHARE

The OpenMP PARALLEL WORKSHARE Fortran directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct. The END PARALLEL WORKSHARE directive is optional.

Syntax:

```
!$OMP PARALLEL WORKSHARE [CLAUSES]
   < Fortran structured block to be executed in parallel >
[!$OMP END PARALLEL WORKSHARE]
```

```
!$OMP PARALLEL DO [CLAUSES]
   < Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)
SCHEDULE (type [, chunk])
COLLAPSE (n)
ORDERED

Usage:

The OpenMP PARALLEL WORKSHARE directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct. The semantics of the PARALLEL WORKSHARE directive are identical to those of a parallel region containing a single WORKSHARE construct.

The END PARALLEL WORKSHARE directive is optional, and NOWAIT may not be specified on an END PARALLEL WORKSHARE directive. The available clauses are as defined in "PARALLEL ... END PARALLEL and parallel ," on page 294.

SECTIONS … END SECTIONS and sections

The OpenMP SECTIONS / END SECTIONS directive pair and the omp sections pragma define a non-iterative work-sharing construct within a parallel region in which each section is executed by a single processor.

Syntax:

```
!$OMP SECTIONS [ Clauses ]
[!$OMP SECTION]
   < Fortran code block executed
  by processor i >
[!$OMP SECTION]
   < Fortran code block executed
  by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

```
#pragma omp sections [ Clauses ]
{
 [#pragma omp section]
   < C/C++ structured block executed
   by processor i >
 [#pragma omp section]
   < C/C++ structured block executed
   by processor j >
 ...
}
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	lastprivate (list)
REDUCTION({operator | intrinsic} : list)	reduction({operator: list)
	nowait

Usage:

The SECTIONS / END SECTIONS directive pair and the omp sections pragma define a non-iterative work-sharing construct within a parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors have no work and thus jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors must execute more than one section.

A SECTION directive or omp sections pragma may only appear within the lexical extent of the enclosing SECTIONS / END SECTIONS directives or omp sections pragma. In addition, the code within the SECTIONS / END SECTIONS directives or omp sections pragma must be a structured block.

The available clauses are as defined in "PARALLEL ... END PARALLEL and parallel ," on page 294 and "DO...END DO and for ".

SINGLE ... END SINGLE and single

The SINGLE...END SINGLE directive or omp single pragma designate code that executes on a single thread and that is skipped by the other threads.

Syntax:

```
!$OMP SINGLE [Clauses]
   < Fortran code executed in body of
  SINGLE processor section >
!$OMP END SINGLE [NOWAIT]
```

```
#pragma omp single [clauses]
   < C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
COPYPRIVATE(list)	copyprivate (list)
	nowait

Usage:

In a parallel region of code, there may be a sub-region of code that only executes correctly on a single thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the SINGLE...END SINGLE directive pair lets you conveniently designate code that executes on a single thread and is skipped by the other threads. There is an implied barrier on exit from a SINGLE...END SINGLE section of code unless the optional NOWAIT clause is specified.

Nested single process sections are ignored. Branching into or out of a single process section is not supported.

Examples:

For Directives:

```
   PROGRAM SINGLE_USE
   INTEGER A(0:1)
   INTEGER omp_get_thread_num()
!$OMP PARALLEL
   A(omp_get_thread_num()) =
     omp_get_thread_num()
!$OMP SINGLE
   PRINT *, "YOU ONLY SEE THIS ONCE"
!$OMP END SINGLE
!$OMP END PARALLEL
   PRINT *,"A(0)=",A(0),
          " A(1)=", A(1)
   END
```

For Pragmas:

```
single_use() {
    int a[2]={0,1};
#pragma omp parallel
{
      a(omp_get_thread_num()) =
     omp_get_thread_num();
#pragma omp single
{
   printf("You only see this once"
} /* end single */
} /* end parallel */
printf("a[0] = %d, a[1] = %d",
        a[0], a[1]);
} /* end single_use */
```

TASK and task

The OpenMP TASK directive and the omp task pragma define an explicit task.

Syntax:

```
!$OMP TASK [Clauses]
< Fortran code executed as task>
!$OMP END TASK
```

```
#pragma omp task [clauses]
< C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
IF(scalar_logical_expression)	if (scalar_expression)
UNTIED	untied
DEFAULT(private | firstprivate | shared | none)	default(shared | none)
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
SHARED(list)	shared(list)

Usage:

The TASK / END TASK directive pair and the omp task pragma define an explicit task.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply. The encountering thread may immediately execute the task, or delay its execution. If the task execution is delayed, then any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs.

A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

When an if clause is present on a task construct and the if clause expression evaluates to false, the encountering thread must suspend the current task region and begin execution of the generated task

immediately, and the suspended task region may not be resumed until the generated task is completed. The task still behaves as a distinct task region with respect to data environment, lock ownership, and synchronization constructs.

Note

Use of a variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs.

A thread that encounters a task scheduling point within the task region may temporarily suspend the task region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the untied clause is present on a task construct, any thread in the team can resume the task region after a suspension.

The task construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit task region includes a task scheduling point at its point of completion. An implementation may add task scheduling points anywhere in untied task regions.

Note

When storage is shared by an explicit task region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

Restrictions:

The following restrictions apply to the TASK directive or omp task pragma:

- A program that branches into or out of a task region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the task directive, or on any side effects of the evaluations of the clauses.
- At most one *if* clause can appear on the directive.
- Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has unspecified behavior.
- In C/C++, a throw executed inside a task region must cause execution to resume within the same task region, and the same thread that threw the exception must catch it.

TASKWAIT and taskwait

The OpenMP TASKWAIT directive and the omp taskwait pragma specify a wait on the completion of child tasks generated since the beginning of the current task.

Syntax:

```
!$OMP TASKWAIT
```

```
#pragma omp taskwait >
```

Clauses:

For Directives:	For Pragmas:
IF(scalar_logical_expression)	if (scalar_expression)
UNTIED	untied
DEFAULT(private | firstprivate | shared | none)	default(shared | none)
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
SHARED(list)	shared(list)

Usage:

The OpenMP TASKWAIT directive and the omp taskwait pragma specify a wait on the completion of child tasks generated since the beginning of the current task.

Restrictions:

The following restrictions apply to the TASKWAIT directive or omp taskwait pragma:

- The TASKWAIT directive and the omp taskwait pragma may be placed only at a point where a base language statement is allowed.
- The taskwait directive may not be used in place of the statement following an *if*, *while*,*do*, *switch*, or *label*.

THREADPRIVATE and threadprivate

The OpenMP THREADPRIVATE directive identifies a Fortran common block as being private to each thread. The omp threadprivate pragma identifies a global variable as being private to each thread.

Syntax:

```
!$OMP THREADPRIVATE (list)
```

```
#pragma omp threadprivate (list)
```

Usage:

The `list` is a list of variables to be made private to each thread but global within the thread. For directives, common block names must appear between slashes, such as `/common_block_name/`.

This directive or pragma must appear in the declarations section of a program unit after the declaration of any variables listed. On entry to a parallel region, data in a threadprivate variable is undefined unless copyin is specified on the parallel directive or pragma. When a variable appears in an threadprivate directive or pragma, each thread's copy is initialized once at an unspecified point prior to its first use as the master copy would be initialized in a serial execution of the program.

Restrictions:

The following restrictions apply to the THREADPRIVATE directive or omp threadprivate pragma:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- The omp threadprivate pragma must appear after the declaration of every threadprivate variable included in list.

- Only named common blocks can be made thread private.
- It is illegal for a THREADPRIVATE common block or its constituent variables to appear in any clause other than a COPYIN clause.
- A variable can appear in a THREADRIVATE directive only in the scope in which it is declared. It must not be an element of a common block or be declared in an EQUIVALENCE statement.
- A variable that appears in a THREADPRIVATE directive and is not declared in the scope of a module must have the SAVE attribute.
- If a variable is specified in an omp threadprivate pragma in one translation unit, it must be specified in an omp threadprivate pragma in every translation unit in which it appears.
- The address of an omp threadprivate variable is not an address constant.
- An omp threadprivate variable must not have an incomplete type or a reference type.

WORKSHARE ... END WORKSHARE

The OpenMP WORKSHARE … END WORKSHARE Fortran directive pair provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

Syntax:

```
!$OMP WORKSHARE
   < Fortran structured block to be executed in parallel >
!$OMP END WORKSHARE [NOWAIT]
```

Usage:

The Fortran structured block enclosed by the WORKSHARE … END WORKSHARE directive pair can consist only of the following types of statements and constructs:

- Array assignments
- Scalar assignments
- FORALL statements or constructs
- WHERE statements or constructs
- OpenMP ATOMIC, CRITICAL or PARALLEL constructs

The work implied by these statements and constructs is split up between the threads executing the WORKSHARE construct in a way that is guaranteed to maintain standard Fortran semantics. The goal of the WORKSHARE construct is to effect parallel execution of non-iterative but implicitly data parallel array assignments, FORALL, and WHERE statements and constructs intrinsic to the Fortran language beginning with Fortran 90. The Fortran structured block contained within a WORKSHARE construct must not contain any user-defined function calls unless the function is ELEMENTAL.

Directive and Pragma Clauses

Some directives and C/C++ pragmas accept clauses that further allow a user to control the scope attributes of variables for the duration of the directive or pragma. Not all clauses are allowed on all directives or pragmas,

so the clauses that are valid are included with the description of the directive or pragma. Typically, if no data scope clause is specified for variables, the default scope is *shared*.

Table 5.2, "Directive and Pragma Clauses Summary Table ," on page 62 provides a brief summary of the clauses associated with OpenMP directives and pragmas that PGI supports. This section contains more information about each of these clauses. For complete information and more details related to use of these clauses, refer to the OpenMP documentation available on the WorldWide Web.

COLLAPSE (n)

The COLLAPSE(n) clause specifies how many loops are associated with the loop construct.

The parameter of the collapse clause must be a constant positive integer expression. If no COLLAPSE clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space, which is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If the loop directive contains a COLLAPSE clause then there may be more than one associated loop.

COPYIN (list)

The COPYIN(list) clause allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region; that is, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

The COPYIN clause applies only to THREADPRIVATE common blocks. If you specify a COPYIN clause, here are a few tips:

- You cannot specify the same entity name more than once in the list.
- You cannot specify the same entity name in separate COPYIN clauses of the same directive.
- You cannot specify both a common block name and any variable within that same named common block in the list.
- You cannot specify both a common block name and any variable within that same named common block in separate COPYIN clauses of the same directive.

COPYPRIVATE(list)

The COPYPRIVATE(list) clause specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.

You use a COPYPRIVATE(list) clause on an END SINGLE directive to cause the variables in the list to be copied from the private copies in the single thread that executes the SINGLE region to the other copies in all other threads of the team at the end of the SINGLE region.

Note

The COPYPRIVATE clause must not appear on the same END SINGLE directive as a NOWAIT clause.

The compiler evaluates a COPYPRIVATE clause before any threads have passed the implied BARRIER directive at the end of that construct.

DEFAULT

The DEFAULT clause specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables. The DEFAULT clause lets you specify the default attribute for variables in the lexical extent of the parallel region. Individual clauses specifying PRIVATE, SHARED, and so on, override the declared DEFAULT.

Specifying DEFAULT(NONE) declares that there is no implicit default. With this declaration, each variable in the parallel region must be explicitly listed with an attribute of PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

FIRSTPRIVATE(list)

The FIRSTPRIVATE(list) clause specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.

Variables that appear in the list of a FIRSTPRIVATE clause are subject to the same semantics as PRIVATE variables; however, these variables are initialized from the original object that exists prior to entering the parallel region.

If a directive construct contains a FIRSTPRIVATE argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

IF()

The IF() clause specifies whether a loop should be executed in parallel or in serial.

In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to .TRUE.. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

LASTPRIVATE(list)

The LASTPRIVATE(list) clause specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).

NOWAIT

The NOWAIT clause overrides the barrier implicit in a directive. When you specify NOWAIT, it removes the implicit barrier synchronization at the end of a for or sections construct.

NUM_THREADS

The NUM_THREADS clause sets the number of threads in a thread team. The num_threads clause allows a user to request a specific number of threads for a parallel construct. If the num_threads clause is present, then

ORDERED

The ORDERED clause specifies that a loop is executed in the order of the loop iterations. This clause is required on a parallel FOR statement when an ordered directive is used in the loop.

You use this clause in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code. If ORDERED constructs are contained in the dynamic extent of the DO construct, the ordered clause must be present on the DO directive.

PRIVATE

The PRIVATE clause specifies that each thread should have its own instance of a variable. Therefore, variables specified in a PRIVATE list are private to each thread in a team. In effect, the compiler creates a separate copy of each of these variables for each thread in the team. When an assignment to a private variable occurs, each thread assigns to its local copy of the variable. When operations involving a private variable occur, each thread performs the operations using its local copy of the variable.

Tips about private variables:

- Variables declared private in a parallel region are undefined upon entry to the parallel region. If the first use of a private variable within the parallel region is in a right-hand-side expression, the results of the expression will be undefined, indicating the probability of a coding error.
- Variables declared private in a parallel region are undefined when serial execution resumes at the end of the parallel region.

REDUCTION

The REDUCTION clause specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. updates named variables declared on the clause within the directive construct.

Intermediate values of REDUCTION variables are not used within the parallel construct, other than in the updates themselves.Variables that appear in the list of a REDUCTION clause must be SHARED. A private copy of each variable in `list` is created for each thread as if the PRIVATE clause had been specified. Each private copy is initialized according to the operator as specified in the following table:

Table 17.1. Initialization of REDUCTION Variables

For Directives		For Pragmas	
Operator / Intrinsic	**Initialization**	**Operator**	**Initialization**
+	0	+	0

For Directives	
Operator / Intrinsic	**Initialization**
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Smallest Representable Number
MIN	Largest Representable Number
IAND	All bits on
IOR	0
IEOR	0

For Pragmas	
Operator	**Initialization**
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

At the end of the parallel region, a reduction is performed on the instances of variables appearing in `list` using operator or intrinsic as specified in the REDUCTION clause. The initial value of each REDUCTION variable is included in the reduction operation. If the `{operator | intrinsic}:` portion of the REDUCTION clause is omitted, the default reduction operator is "+" (addition).

SCHEDULE

The SCHEDULE clause specifies how iterations of the DO or for loop are divided up between processors. Given a SCHEDULE (type [, chunk]) clause, the type can be STATIC, DYNAMIC, GUIDED, or RUNTIME, defined in the following list.

Note

For pragmas, the values for the clause are lower case static, dynamic, guided, or runtime. For simplicity, we use the directive uppercase value in the following information.

- When SCHEDULE (STATIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. The blocks of iterations are statically assigned to threads in a round-robin fashion in order of the thread ID numbers. The chunk must be a scalar integer expression. If chunk is not specified, a default chunk size is chosen equal to:

```
(number_of_iterations + omp_num_threads() - 1) / omp_num_threads()
```

- When SCHEDULE (DYNAMIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. The chunk must be a scalar integer expression. If no chunk is specified, a default chunk size is chosen equal to 1.
- When SCHEDULE (GUIDED, chunk) is specified, the chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. Chunk specifies the minimum number of

iterations to dispatch each time, except when there are less than chunk iterations remaining to be processed, at which point all remaining iterations are assigned. If no chunk is specified, a default chunk size is chosen equal to 1.

- When SCHEDULE (RUNTIME) is specified, the decision regarding iteration scheduling is deferred until runtime. The schedule type and chunk size can be chosen at runtime by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is equivalent to SCHEDULE(STATIC).

SHARED

The SHARED clause specifies variables that must be available to all threads. If you specify a variable as SHARED, you are stating that all threads can safely share a single copy of the variable. When one or more variables are shared among all threads, all threads access the same storage area for the shared variables.

UNTIED

The UNTIED clause specifies that any thread in the team can resume the task region after a suspension.

Note

The thread number may change at any time during the execution of an untied task. Therefore, the value returned by `omp_get_thread_num` is generally not useful during execution of such a task region.

OpenMP Environment Variables

OpenMP environment variables allow you to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives and pragmas.

Table 5.4, "OpenMP-related Environment Variable Summary Table," on page 71 provides a brief summary of these variables. This section contains more information about each of them. For complete information and more details related to these environment variables, refer to the OpenMP documentation available on the WorldWide Web.

OMP_DYNAMIC

`OMP_DYNAMIC` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.

OMP_NESTED

`OMP_NESTED` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) nested parallelism.

OMP_MAX_ACTIVE_LEVELS

`OMP_MAX_ACTIVE_LEVELS` currently has no effect. Typically this variable specifies the maximum number of nested parallel regions. PGI ignores this variable value since nested parallelism is not supported.

OMP_NUM_THREADS

OMP_NUM_THREADS specifies the number of threads to use during execution of parallel regions. The default value for this variable is 1. For historical reasons, the environment variable NCPUS is supported with the same functionality. In the event that both OMP_NUM_THREADS and NCPUS are defined, the value of OMP_NUM_THREADS takes precedence.

Note

OMP_NUM_THREADS defines the threads that are used to execute the program, regardless of the number of physical processors available in the system. As a result, you can run programs using more threads than physical processors and they execute correctly. However, performance of programs executed in this manner can be unpredictable, and oftentimes will be inefficient.

OMP_SCHEDULE

OMP_SCHEDULE specifies the type of iteration scheduling to use for DO and PARALLEL DO loop directives and for omp for and omp parallel for loop pragmas that include the SCHEDULE(RUNTIME) clause, described in "SCHEDULE," on page 307. The default value for this variable is STATIC.

If the optional chunk size is not set, a chunk size of 1 is assumed except in the case of a static schedule. For a static schedule, the default is as defined in "DO...END DO and for ," on page 290.

Examples of the use of OMP_SCHEDULE are as follows:

For Fortran:

```
% setenv OMP_SCHEDULE "STATIC, 5"
% setenv OMP_SCHEDULE "GUIDED, 8"
% setenv OMP_SCHEDULE "DYNAMIC"
```

For C/C++:

```
% setenv OMP_SCHEDULE "static, 5"
% setenv OMP_SCHEDULE "guided, 8"
% setenv OMP_SCHEDULE "dynamic"
```

OMP_STACKSIZE

OMP_STACKSIZE is an OpenMP 3.0 feature that controls the size of the stack for newly-created threads. This variable overrides the default stack size for a newly created thread. The value is a decimal integer followed by an optional letter B, K, M, or G, to specify bytes, kilobytes, megabytes, and gigabytes, respectively. If no letter is used, the default is kilobytes. There is no space between the value and the letter; for example, one megabyte is specified 1M. The following example specifies a stack size of 8 megabytes.

```
% setenv OMP_STACKSIZE 8M
```

The API functions related to OMP_STACKSIZE are omp_set_stack_size and omp_get_stack_size.

The environment variable OMP_STACKSIZE is read on program start-up. If the program changes its own environment, the variable is not re-checked.

This environment variable takes precedence over MPSTKZ, described in "MPSTKZ," on page 132. Once a thread is created, its stack size cannot be changed.

In the PGI implementation, threads are created prior to the first parallel region and persist for the life of the program. The stack size of the main thread (thread 0) is set at program start-up and is not affected by

OMP_STACKSIZE. For more information on controlling the program stack size in Linux, refer to "Running Parallel Programs on Linux," on page 9.

OMP_THREAD_LIMIT

You can use the OMP_THREAD_LIMIT environment variable to specify the absolute maximum number of threads that can be used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using set_omp_num_threads(), cause the number of processes or threads to be set at the value of OMP_THREAD_LIMIT rather than the value specified in the function call.

OMP_WAIT_POLICY

OMP_WAIT_POLICY sets the behavior of idle threads - specifically, whether they spin or sleep when idle. The values are ACTIVE and PASSIVE, with ACTIVE the default. The behavior defined by OMP_WAIT_POLICY is also shared by threads created by auto-parallelization.

- Threads are considered idle when waiting at a barrier, when waiting to enter a critical region, or when unemployed between parallel regions.
- Threads waiting for critical sections always busy wait (ACTIVE).
- Barriers always busy wait (ACTIVE), with calls to sched_yield determined by the environment variable MP_SPIN, described in "MP_SPIN," on page 133.
- Unemployed threads during a serial region can either busy wait using the barrier (ACTIVE) or politely wait using a mutex (PASSIVE). This choice is set by OMP_WAIT_POLICY, so the default is ACTIVE.

When ACTIVE is set, idle threads consume 100% of their CPU allotment spinning in a busy loop waiting to restart in a parallel region. This mechanism allows for very quick entry into parallel regions, a condition which is good for programs that enter and leave parallel regions frequently.

When PASSIVE is set, idle threads wait on a mutex in the operating system and consume no CPU time until being restarted. Passive idle is best when a program has long periods of serial activity or when the program runs on a multi-user machine or otherwise shares CPU resources.

Chapter 18. PGI Accelerator Compilers Reference

Chapter 7, "*Using an Accelerator*" describes the programming model that uses a collection of compiler directives to specify regions of code in Fortran and C programs that can be offloaded from a *host* CPU to an attached *accelerator*. The method described provides a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators.

PGI provides a set of Fortran and C accelerator compilers and tools for 64-bit x86-compatible processor-based workstations and servers running 64-bit Linux operating systems, with an attached NVIDIA CUDA-enabled GPU or Tesla card.

Note

The PGI Accelerator compilers require a separate license key in addition to a normal PGI Workstation, Server, or CDK license.

This chapter contains detailed descriptions of each of the PGI Accelerator directives and C pragmas that PGI supports. In addition, the section "PGI Accelerator Directive Clauses," on page 314 contains information about the clauses associated with these directives and pragmas.

PGI Accelerator Directives

This section provides detailed descriptions of the Fortran and C directives used to delineate accelerator regions and to augment information available to the compiler for scheduling of loops and classification of data.

- In C, PGI Accelerator directives are specified using the #pragma mechanism provided by the standard.
- In Fortran, PGI Accelerator directives are specified using special comments that are identified by a unique sentinel.

This syntax enables compilers to ignore accelerator directives if support is disabled or not provided.

PGI currently supports three types of accelerator directives:

- An "Accelerator Region Directive" defines the region of a program to be compiled for execution on the accelerator device.

- An "Accelerator Loop Mapping Directive" describes the type of parallelism to use to execute the loop and declare loop-private variables and arrays.
- An "Combined Directives" is a combination of the Accelerator region and loop mapping directive, and specifies a loop directive nested immediately inside an accelerator region directive.

Accelerator Region Directive

This directive defines the region of the program that should be compiled for execution on an accelerator device.

Syntax

In C, the syntax of an accelerator region directive is:

```
#pragma acc region [clause [, clause]…] new-line
 structured block
```

In Fortran, the syntax is:

```
!$acc region [clause [, clause]…]
    structured block
!$acc end region
```

where clause is one of the following:

```
if( condition )
copy( list )
copyin( list )
copyout( list )
local( list )
```

Description

Loops within the structured block are compiled into accelerator kernels. Data is copied from the host memory to the accelerator memory, as required, and result data is copied back. Any computation that cannot be executed on the accelerator, perhaps because of limitations of the device, is executed on the host. This approach may require data to move back and forth between the host and device.

At the end of the region, all results stored on the device that are needed on the host are copied back to the host memory, and accelerator memory is deallocated.

Restrictions

The following restrictions apply to the accelerator region directive:

- Accelerator regions may not be nested.
- A program may not branch into or out of an accelerator region.
- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one `if` clause may appear. In Fortran, the condition must evaluate to a scalar logical value; in C, the condition must evaluate to a scalar integer value.

- A variable may appear in only one of the `local`, `copy`, `copyin` or `copyout` lists.

Accelerator Loop Mapping Directive

An accelerator loop mapping directive specifies the type of parallelism to use to execute the loop and declare loop-private variables and arrays.

Syntax

In C, the syntax of an accelerator loop mapping directive is

```
      #pragma acc for [clause [,clause]…] new-line
           for loop
```

In Fortran, the syntax of an accelerator loop mapping directive is

```
  !$acc do [clause [,clause]…]
           do loop
```

where clause is one of the following:

```
host [(width)]
parallel [(width)]
seq [(width)]
vector [(width)]
kernel
private( list )
```

Description

An accelerator loop mapping directive applies to a loop which must appear on the following line. It can describe what type of parallelism to use to execute the loop and declare loop-private variables and arrays.

Combined Directives

The combined accelerator region and loop mapping directive is a shortcut for specifying a loop directive nested immediately inside an accelerator region directive. The meaning is identical to explicitly specifying a region construct containing a loop directive. Any clause that is allowed on a region directive or a loop directive is allowed on a combined directive.

Syntax

In C, the syntax of the combined accelerator region and loop directive is:

```
 #pragma acc region for [clause [, clause]...] new-line
             for loop
```

In Fortran the syntax of the combined accelerator region and loop directive is:

```
   !$acc region do [clause [, clause]...]
             do loop
```

where *clause* is any of the region or loop clauses described previously in this chapter.

The associated region is the body of the loop which must immediately follow the directive.

Restrictions

The following restrictions apply to a combined directive:

- The combined accelerator region and loop directive may not appear within the body of another accelerator region.
- All restrictions for the region directive and the loop directive apply.

PGI Accelerator Directive Clauses

Accelerator directives accept clauses that further clarify the use of the directive. Some of these clauses are specific to certain directives. The following

Accelerator Region Directive Clauses

The following clauses further clarify the use of the Accelerator Region directive.

if clause

The `if` clause is optional.

- When there is no `if` clause, the compiler generates code to execute as much of the region on the accelerator as possible.
- When an `if` clause appears, the compiler generates two copies of the region, one copy to execute on the accelerator and one copy to execute on the host. The condition in the `if` clause determines whether the host or accelerator copy is executed.
  - When the condition in the `if` clause evaluates to zero in C, or .false. in Fortran, the host copy is executed.
  - When the condition in the `if` clause evaluates to nonzero in C, or .true. in Fortran, the accelerator copy is executed.

Data Clauses

The data clauses for an accelerator region directive are one of the following:

```
copy( list )
copyout( list )
copyin( list )
local( list )
```

Data clauses are optional, but may assist the compiler in generating code for an accelerator or in generating more optimal accelerator kernels.

Note

By default, the PGI Accelerator compilers attempt to minimize data movement between the host and accelerator. As a result, for many accelerator regions the compilers choose to copy sub-arrays which may be non-contiguous. Performance of an accelerator may improve in these cases if the user

inserts explicit copy/copyin/copyout clauses on the accelerator region directive to specify to copy whole arrays rather than sub-arrays. Depending on the architecture of the target accelerator memory, performance also may improve if one or more dimensions of copied arrays are padded.

For each variable or array used in the region that does not appear in any data clause, the compiler analyzes all references to the variable or array and determines:

- For arrays, how much memory needs to be allocated in the accelerator memory to hold the array;
- Whether the value in host memory needs to be copied to the accelerator memory;
- Whether a value computed on the accelerator will be needed again on the host, and therefore needs to be copied back to the host memory.

When compiler analysis is unable to determine these items, it may fail to generate code for the accelerator; in that case, it issues a message to notify the programmer why it failed. You can use data clauses to augment or override this compiler analysis.

When a data clause is used, the list argument is a comma-separated collection of variable names, array names, or subarray specifications.

Using an array name without bounds tells the compiler to use the whole array. Thus, every array reference is equivalent to some subarray of that array.

- Using an array name without bounds tells the compiler to use the whole array. Thus, every array reference is equivalent to some subarray of that array.
- In C, a subarray is an array name followed by a range specification in brackets, such as the following:

```
arr[2:high]
```

- In Fortran, a subarray is an array name followed by a comma-separated list of range specifications in parentheses, such as the following:

```
arr(2:high,low:100)
```

Data clause restrictions

An accelerator region data clause has these restrictions:

- If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if known, are used.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.
- If a subarray is specified, then only that subarray of the array needs to be copied.
- Only one subarray for an array may appear in any data clause for a region.
- The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.

copy clause

You use the `copy` clause to declare that the variables, arrays or subarrays in the list have values in the host memory that need to be copied to the accelerator memory, and are assigned values on the accelerator that need to be copied back to the host.

copyin clause

You use the `copyin` clause to declare that the variables, arrays or subarrays in the list have values in the host memory that need to be copied to the accelerator memory.

copyout clause

You use the `copyout` clause to declare that the variables, arrays, or subarrays in the list are assigned or contain values in the accelerator memory that need to be copied back to the host memory at the end of the accelerator region.

local clause

You use the `local` clause to declare that the variables, arrays or subarrays in the list need to be allocated in the accelerator memory, but the values in the host memory are not needed on the accelerator, and the values computed and assigned on the accelerator are not needed on the host.

loop Directive Clauses

The loop scheduling clauses tell the compiler about loop level parallelism and how to map the parallelism onto the accelerator parallelism.

The loop scheduling clauses for the accelerator loop mapping directive are one of the following:

```
host [(width)]
parallel [(width)]
seq [(width)]
vector [(width)]
```

The loop scheduling clauses tell the compiler about loop level parallelism and how to map the parallelism onto the accelerator parallelism.

The loop scheduling clauses are optional.

For each loop without a scheduling clause, the compiler determines an appropriate schedule automatically.

loop scheduling clauses restrictions

The loop scheduling clauses have these restrictions:

- In some cases, there is a limit on the trip count of a parallel loop on the accelerator. For instance, some accelerators have a limit on the maximum length of a vector loop. In such cases, the compiler strip-mines the loop, so one of the loops has a maximum trip count that satisfies the limit.

  For example, if the maximum vector length is 256, the compiler uses strip-mining to compile a vector loop like the following one:

```
!$acc do vector
  do i = 1,n
```

into the following pair of loops:

```
do is = 1,n,256
 !$acc do vector
   do i = is,max(is+255,n)
```

The compiler then chooses an appropriate schedule for the outer, strip loop.

- If more than one scheduling clause appears on the loop directive, the compiler strip-mines the loop to get at least that many nested loops, applying one loop scheduling clause to each level.
- If a loop scheduling clause has a width argument, the compiler strip-mines the loop to that width, applying the scheduling clause to the outer strip or inner element loop, and then determines the appropriate schedule for the other loop.
- The width argument must be a compile-time positive constant integer.
- If two or more loop scheduling clauses appear on a single loop mapping directive, all but one must have a width argument.
- Some implementations or targets may require the width expression for the vector clause to be a compile-time constant.
- Some implementations or targets may require the width expression for the vector or parallel clauses to be a power of two, or a multiple of some power of two. If so, the behavior when the restriction is violated is implementation-defined.

loop scheduling clause examples

In the following example, the compiler strip-mines the loop to 16 host iterations:

```
!$acc do host(16), parallel
  do i = 1,n
```

The parallel clause applies to the inner loop, as follows:

```
ns = ceil(n/16)
!$acc do host
  do is = 1, n, ns
  !$acc do parallel
    do i = is, min(n,is+ns-1)
```

host clause

The `host` clause tells the compiler to execute the loop sequentially on the host processor. There is no maximum number of iterations on a host schedule. If the `host` clause has a width argument, the compiler strip mines the loop to that many strips, and determines an appropriate schedule for the remaining loop.

parallel clause

The `parallel` clause tells the compiler to execute this loop in parallel mode on the accelerator. There may be a target-specific limit on the number of iterations in a parallel loop or on the number of parallel loops allowed in a given kernel. If there is a limit:

- If there is no width argument, or the value of the width expression is greater than the limit, the compiler enforces the limit.
- If there is a width argument or a limit on the number of iterations in a parallel loop, then only that many iterations can run in parallel at a time.

seq clause

The `seq` clause tells the compiler to execute this loop sequentially on the accelerator. There is no maximum number of iterations for a seq schedule. If the `seq` clause has a width argument, the compiler strip mines the loop and determines an appropriate schedule for the remaining loop.

vector clause

The `vector` clause tells the compiler to execute this loop in vector mode on the accelerator. There may be a target-specific limit on the number of iterations in a vector loop, the aggregate number of iterations in all vector loops, or the number of vector loops allowed in a kernel.

When there is a limit:

- If there is no width argument, or the value of the width expression is greater than the limit, the compiler strip mines the loop to enforce the limit.

kernel clause

The `kernel` clause tells the compiler that the body of this loop is to be the body of the computational kernel. Any loops contained within the kernel loop are executed sequentially on the accelerator.

kernel clause restrictions

The `kernel` clause has this restriction:

- Loop mapping directives must not appear on any loop contained within the kernel loop.

private clause

You use the `private` clause to declare that the variables, arrays, or subarrays in the list argument need to be allocated in the accelerator memory with one copy for each iteration of the loop.

Any value of the variable or array used in the loop must have been computed and assigned in that iteration of the loop, and the values computed and assigned in any iteration are not needed after completion of the loop.

Using an array name without bounds tells the compiler to analyze the references to the array to determine what bounds to use. If the lower or upper bounds are missing, the declared or allocated bounds, if known, are used.

private clause restrictions

The `private` clause has these restrictions:

- A variable, array or subarray may only appear once in any `private` clause for a loop.

- Only one subarray for an array may appear in any `private` clause for a loop.
- If a subarray appears in a `private` clause, then the compiler only needs to allocate memory to hold that subarray in the accelerator memory.
- The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.
- If a subarray appears in a `private` clause, it is an error to refer to any element of the array in the loop outside the bounds of the subarray.
- It is an error to refer to a variable or any element of an array or subarray that appears in a `private` clause and that has not been assigned in this iteration of the loop.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.

PGI Accelerator Runtime Routines

This section defines specific details related to user-callable functions and library routines that are available for use by programmers to query the accelerator features and to control behavior of accelerator-enabled programs at runtime.

acc_get_device

The `acc_get_device` routine returns the type of accelerator device being used.

Syntax

In C, the syntax is this:

```
int acc_get_device(void);
```

In Fortran, the syntax is this:

```
integer function acc_get_device()
```

Description

The `acc_get_device` routine returns the type of accelerator device being used. Its return value is one of the predefined values in accel.h, accel_lib.h, or the accel_lib module. It may not be called within an accelerator region.

acc_get_num_devices

The acc_get_num_devices routine returns the number of accelerator devices of the given type attached to the host.

Syntax

In C, the syntax is this:

```
int acc_get_num_devices(acc_device_t);
```

In Fortran, the syntax is this:

```
integer function acc_get_num_devices(devicetype)
integer(acc_device_kind) devicetype
```

Description

The `acc_get_num_devices` routine returns the number of accelerator devices of the given type attached to the host.

The argument determines what kind of device to count. The possible values for the argument are implementation-specific, and are listed in the C include file `accel.h`, the Fortran include file `accel_lib.h` and the Fortran module `accel_lib`.

acc_init

The `acc_init` routine connects to and initializes the accelerator device and allocates the control structures in the accelerator library.

Syntax

In C, the syntax is this:

```
int acc_init(acc_device_t);
```

In Fortran, the syntax is this:

```
subroutine acc_init( devicetype )
  integer(acc_device_kind) devicetype
```

Description

The `acc_init` routine connects to and initializes the accelerator device and allocates the control structures in the accelerator library. The `acc_init` routine must be called before entering any accelerator regions. Its argument must be one of the predefined values in `accel.h`, `accel_lib.h` or the `accel_lib` module.

acc_set_device_num

The `acc_set_device_num` routine tells the runtime which device to use when executing an accelerator region.

Syntax

In C, the syntax is this:

```
int acc_set_device_num(int, acc_device_t);
```

In Fortran, the syntax is this:

```
subroutine acc_set_device_num( devicenum, devicetype )
   integer devicenum
  integer(acc_device_kind) devicetype
```

Description

The `acc_set_device_num` routine tells the runtime which device to use among those attached of the given type.

- If the value of devicenum is zero, the runtime reverts to its default behavior, which is implementation-defined.
- If the value of devicenum is greater than the value returned by `acc_get_num_devices` for that device type, the behavior is implementation-defined.
- If the value of the second argument is zero, the selected device number is used for all attached accelerator types.

Note

This routine may not be called during execution of an accelerator region.

Environment Variables

This section describes the environment variables that PGI supports to control behavior of accelerator-enabled programs at execution.

ACC_DEVICE

When a program is compiled with the PGI Unified Binary, the ACC_DEVICE environment variable controls the default device to use when executing a program. The value of this environment variable must be set to `NVIDIA` or `nvidia`, indicating to run on the NVIDIA GPU. Currently, any other value of the environment variable causes the program to use the host version.

Example

The following example indicates to use the NVIDIA GPU when executing the program:

```
setenv ACC_DEVICE nvidia
export ACC_DEVICE=nvidia
```

ACC_DEVICE_NUM

The ACC_DEVICE_NUM environment variable controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host.

- If the value is zero, the implementation-defined default is used.
- If the value is greater than the number of devices attached, the behavior is implementation-defined.

Example

The following example indicates how to set the default device number to use when executing accelerator regions:

```
setenv ACC_DEVICE_NUM 1
```

```
export ACC_DEVICE_NUM=1
```

ACC_NOTIFY

The ACC_NOTIFY environment variable, when set to a non-negative integer, indicates to print a message for each kernel launched on the device.

Example

The following example indicates to print a message for each kernel launched on the device:

```
setenv ACC_NOTIFY 1
export ACC_NOTIFY=1
```

Chapter 19. C++ Name Mangling

Name mangling transforms the names of entities so that the names include information on aspects of the entity's type and fully qualified name. This ability is necessary since the intermediate language into which a program is translated contains fewer and simpler name spaces than there are in the C++ language; specifically:

- Overloaded function names are not allowed in the intermediate language.
- Classes have their own scopes in C++, but not in the generated intermediate language. For example, an entity x from inside a class must not conflict with an entity x from the file scope.
- External names in the object code form a completely flat name space. The names of entities with external linkage must be projected onto that name space so that they do not conflict with one another. A function f from a class A, for example, must not have the same external name as a function f from class B.
- Some names are not names in the conventional sense of the word, they're not strings of alphanumeric characters, for example: operator=.

There are two main problems here:

1. Generating external names that will not clash.
2. Generating alphanumeric names for entities with strange names in C++.

Name mangling solves these problems by generating external names that will not clash, and alphanumeric names for entities with strange names in C++. It also solves the problem of generating hidden names for some behind-the-scenes language support in such a way that they match up across separate compilations.

You see mangled names if you view files that are translated by PGC++, and you do not use tools that demangle the C++ names. Intermediate files that use mangled names include the assembly and object files created by the pgcpp command. To view demangled names, use the tool pgdecode, which takes input from stdin.

```
prompt> pgdecode
g__1ASFf
A::g(float)
```

The name mangling algorithm for the PGC++ compiler is the same as that for cfront, and, except for a few minor details, also matches the description in Section 8.0, Function Name Encoding, of The Annotated C++ Reference Manual (ARM). Refer to the ARM for a complete description of name mangling.

Types of Mangling

The following entity names are mangled:

- Function names including non-member function names are mangled, to deal with overloading. Names of functions with extern "C" linkage are not mangled.
- Mangled function names have the function name followed by __ followed by F followed by the mangled description of the types of the parameters of the function. If the function is a member function, the mangled form of the class name precedes the F. If the member function is static, an S also precedes the F.

```
int f(float); // f__Ff
class A
 int f(float); // f__1AFf
 static int g(float); // g__1ASFf
;
```

- Special and operator function names, like constructors and operator=(). The encoding is similar to that for normal functions, but a coded name is used instead of the routine name:

```
class A
 int operator+(float); // __pl__1Aff
 A(float); // __ct__1Aff
;
int operator+(A, float); // __pl__F1Af
```

- Static data member names. The mangled form is the member name followed by __ followed by the mangled form of the class name:

```
class A
 static int i; // i__1A
;
```

- Names of variables generated for virtual function tables. These have names like `vtblmangled-class-name` or `vtblmangled-base-class-namemangled-class-name`.
- Names of variables generated to contain runtime type information. These have names like `Ttype-encoding` and `TIDtype-encoding`.

Mangling Summary

This section lists some of the C++ entities that are mangled and provides some details on the mangling algorithm. For more details, refer to The Annotated C++ Reference Manual.

Type Name Mangling

Using PGC++, each type has a corresponding mangled encoding. For example, a class type is represented as the class name preceded by the number of characters in the class name, as in `5abcde` for `abcde`. Simple types are encoded as lower-case letters, as in i for int or f for float. Type modifiers and declarators are encoded as upper-case letters preceding the types they modify, as in U for unsigned or P for pointer.

Nested Class Name Mangling

Nested class types are encoded as a Q followed by a digit indicating the depth of nesting, followed by a _, followed by the mangled-form names of the class types in the fully-qualified name of the class, from outermost to innermost:

```
class A
 class B // Q2_1A1B
 ;
;
```

Local Class Name Mangling

The name of the nested class itself is mangled to the form described previously with a prefix ___, which serves to make the class name distinct from all user names. Local class names are encoded as L followed by a number, followed by __, followed by the mangled name of the class. The number has no special meaning; it's just an identifying number assigned to the class. The name of the class is not in the ARM, and cfront encodes local class names slightly differently.

```
void f()
 class A // L1__1A}
 ;
;
```

This form is used when encoding the local class name as a type. It's not necessary to mangle the name of the local class itself unless it's also a nested class.

Template Class Name Mangling

Template classes have mangled names that encode the arguments of the template:

```
template<class T1, class T2> class abc ;
abc<int, int> x;
abc__pt__3_ii
```

This describes two template arguments of type int with the total length of template argument list string, including the underscore, and a fixed string, indicates parameterized type as well, the name of the class template.

Chapter 20. Directives and Pragmas Reference

As we mentioned in Chapter 8, "*Using Directives and Pragmas*," on page 101, PGI Fortran compilers support proprietary directives and pragmas.

Directives and pragmas override corresponding command-line options. For usage information such as the scope and related command-line options, refer to "*Using Directives and Pragmas*".

This chapter contains detailed descriptions of PGI's proprietary directives and pragmas.

PGI Proprietary Fortran Directive and C/C++ Pragma Summary

Directives are Fortran comments and pragmas are C/C++ comments that the user may supply in a source file to provide information to the compiler. These comments alter the effects of certain command line options or default behavior of the compiler. They provide pragmatic information that control the actions of the compiler in a particular portion of a program without affecting the program as a whole. That is, while a command line option affects the entire source file that is being compiled, directives and pragmas apply, or disable, the effects of a command line option to selected subprograms or to selected loops in the source file, for example, to optimize a specific area of code. Use directives and pragmas to tune selected routines or loops.

As outlined in Chapter 8, "*Using Directives and Pragmas*," on page 101, the Fortran directives may have any of the following forms:

```
!pgi$g directive
!pgi$r directive
!pgi$l directive
!pgi$ directive
```

where the scope indicator follows the $ and is either g (global), r (routine), or l (loop). This indicator controls the scope of the directive, though some directives ignore the scope indicator.

Note

If the input is in fixed format, the comment character, !, * or C, must begin in column 1.

Directives and pragmas override corresponding command-line options. For usage information such as the scope and related command-line options, refer to Chapter 8, "*Using Directives and Pragmas*," on page 101.

altcode (noaltcode)

The `altcode` directive or pragma instructs the compiler to generate alternate code for vectorized or parallelized loops.

The `noaltcode` directive or pragma disables generation of alternate code.

Scope: This directive or pragma affects the compiler only when `-Mvect=sse` or `-Mconcur` is enabled on the command line.

cpgi$ altcode
: Enables alternate code (altcode) generation for vectorized loops. For each loop the compiler decides whether to generate altcode and what type(s) to generate, which may be any or all of: altcode without iteration peeling, altcode with non-temporal stores and other data cache optimizations, and altcode based on array alignments calculated dynamically at runtime. The compiler also determines suitable loop count and array alignment conditions for executing the alternate code.

cpgi$ altcode alignment
: For a vectorized loop, if possible, generates an alternate vectorized loop containing additional aligned moves which is executed if a runtime array alignment test is passed.

cpgi$ altcode [(n)] concur
: For each auto-parallelized loop, generates an alternate serial loop to be executed if the loop count is less than or equal to n. If n is omitted or n is 0, the compiler determines a suitable value of n for each loop.

cpgi$ altcode [(n)] concurreduction
: Sets the loop count threshold for parallelization of reduction loops to n. For each auto-parallelized reduction loop, generate an alternate serial loop to be executed if the loop count is less than or equal to n. If n is omitted or n is 0, the compiler determines a suitable value of n for each loop.

cpgi$ altcode [(n)] nontemporal
: For a vectorized loop, if possible, generates an alternate vectorized loop containing non-temporal stores and other cache optimizations to be executed if the loop count is greater than n. If n is omitted or n is 1, the compiler determines a suitable value of n for each loop. The alternate code is optimized for the case when the data referenced in the loop does not all fit in level 2 cache.

cpgi$ altcode [(n)] nopeel
: For a vectorized loop where iteration peeling is performed by default, if possible, generates an alternate vectorized loop without iteration peeling to be executed if the loop count is less than or equal to n. If n is omitted or n is 1, the compiler determines a suitable value of n for each loop, and in some cases it may decide not to generate an alternate unpeeled loop.

cpgi$ altcode [(n)] vector
: For each vectorized loop, generates an alternate scalar loop to be executed if the loop count is less than or equal to n. If n is omitted or n is 1, the compiler determines a suitable value of n for each loop.

cpgi$ noaltcode
: Sets the loop count thresholds for parallelization of all innermost loops to 0, and disables alternate code generation for vectorized loops.

assoc (noassoc)

This directive or pragma toggles the effects of the `-Mvect=noassoc` command-line option, an optimization `-M` control.

Scope: This directive or pragma affects the compiler only when `-Mvect=sse` is enabled on the command line.

By default, when scalar reductions are present the vectorizer may change the order of operations, such as dot product, so that it can generate better code. Such transformations may change the result of the computation due to roundoff error. The noassoc directive disables these transformations.

bounds (nobounds)

This directive or pragma alters the effects of the `-Mbounds` command line option. This directive enables the checking of array bounds when subscripted array references are performed. By default, array bounds checking is not performed.

cncall (nocncall)

This directive or pragma indicates that loops within the specified scope are considered for parallelization, even if they contain calls to user-defined subroutines or functions. A nocncall directive cancels the effect of a previous cncall.

concur (noconcur)

This directive or pragma alters the effects of the `-Mconcur` command-line option. The directive instructs the auto-parallelizer to enable auto-concurrentization of loops.

Scope: This directive or pragma affects the compiler only when `-Mconcur` is enabled on the command line.

If concur is specified, the compiler uses multiple processors to execute loops which the auto-parallelizer determines to be parallelizable. The noconcur directive disables these transformations; however, use of concur overrides previous noconcur statements.

depchk (nodepchk)

This directive or pragma alters the effects of the `-Mdepchk` command line option. When potential data dependencies exist, the compiler, by default, assumes that there is a data dependence that in turn may inhibit certain optimizations or vectorizations. nodepchk directs the compiler to ignore unknown data dependencies.

eqvchk (noeqvchk)

The eqvchk directive or pragma specifies to check dependencies between EQUIVALENCE associated elements. When examining data dependencies, noeqvchk directs the compiler to ignore any dependencies between variables appearing in `EQUIVALENCE` statements.

fcon (nofcon)

This C/C++ pragma alters the effects of the `-Mfcon` (a –M Language control) command-line option.

The pragma instructs the compiler to treat non-suffixed floating-point constants as float rather than double. By default, all non-suffixed floating-point constants are treated as double.

Note

Only routine or global scopes are allowed for this C/C++ pragma.

invarif (noinvarif)

This directive or pragma has no corresponding command-line option. Normally, the compiler removes certain invariant if constructs from within a loop and places them outside of the loop. The directive noinvarif directs the compiler not to move such constructs. The directive invarif toggles a previous noinvarif.

ivdep

The ivdep directive assists the compiler's dependence analysis and is equivalent to the directive nodepchk.

lstval (nolstval)

This directive or pragma has no corresponding command-line option. The compiler determines whether the last values for loop iteration control variables and promoted scalars need to be computed. In certain cases, the compiler must assume that the last values of these variables are needed and therefore computes their last values. The directive nolstval directs the compiler not to compute the last values for those cases.

prefetch

The prefetch directive or pragma the compiler emits prefetch instructions whereby elements are fetched into the data cache prior to first use. By varying the prefetch distance, it is sometimes possible to reduce the effects of main memory latency and improve performance.

The syntax of this directive or pragma is:

```
c$mem prefetch <var1>[,<var2>[,...]]
```

where `<varn>` is any valid variable, member, or array element reference.

opt

The opt directive or pragma overrides the value specified by the command line option `-On`.

The syntax of this directive or pragma is:

```
cpgi$<scope> opt=<level>
```

where the optional <scope> is *r* or *g* and <level> is an integer constant representing the optimization level to be used when compiling a subprogram (routine scope) or all subprograms in a file (global scope).

safe (nosafe)

This C/C++ pragma has no corresponding command-line option. By default, the compiler assumes that all pointer arguments are unsafe. That is, the storage located by the pointer can be accessed by other pointers.

The formats of the safe pragma are:

```
#pragma [scope] [no]safe
#pragma safe (variable [, variable]...)
```

where scope is either global or routine.

- When the pragma safe is not followed by a variable name or a list of variable names:
  - If the scope is routine, then the compiler treats all pointer arguments appearing in the routine as safe.
  - If the scope is global, then the compiler treats all pointer arguments appearing in all routines as safe.
- When the pragma safe is followed by a variable name or a list of variable names, each name is the name of a pointer argument in the current function, and the compiler considers that named argument to be safe.

  ### Note

  If only one variable name is specified, you may omit the surrounding parentheses.

safe_lastval

During parallelization, scalars within loops need to be privatized. Problems are possible if a scalar is accessed outside the loop. If you know that a scalar is assigned on the last iteration of the loop, making it safe to parallelize the loop, you use the safe_lastval directive or pragma to let the compiler know the loop is safe to parallelize.

For example, use the following C pragma to tell the compiler that for a given loop the last value computed for all scalars make it safe to parallelize the loop:

```
cpgi$l safe_lastval
#pragma loop safe_lastval
```

The command-line option `-Msafe_lastval` provides the same information for all loops within the routines being compiled, essentially providing global scope.

In the following example, a problem results since the value of `t` may not be computed on the last iteration of the loop.

```
do i = 1, N
if( f(x(i)) > 5.0 then)
t = x(i)
endif
enddo
v = t
```

If a scalar assigned within a loop is used outside the loop, we normally save the last value of the scalar. Essentially the value of the scalar on the "last iteration" is saved, in this case when `i=N`.

If the loop is parallelized and the scalar is not assigned on every iteration, it may be difficult to determine on what iteration `t` is last assigned, without resorting to costly critical sections. Analysis allows the compiler to

determine if a scalar is assigned on every iteration, thus the loop is safe to parallelize if the scalar is used later. An example loop is:

```
do i = 1, N
if( x(i) > 0.0 ) then
t = 2.0
else
t = 3.0
endif
y(i) = ...t...
enddo
v = t
```

where `t` is assigned on every iteration of the loop. However, there are cases where a scalar may be privatizable. If it is used after the loop, it is unsafe to parallelize. Examine this loop:

```
do i = 1,N
 if( x(i) > 0.0 ) then
 t = x(i)
 ...
 ...
 y(i) = ...t..
 endif
enddo
v = t
```

where each use of `t` within the loop is reached by a definition from the same iteration. Here `t` is privatizable, but the use of `t` outside the loop may yield incorrect results since the compiler may not be able to detect on which iteration of the parallelized loop `t` is assigned last.

The compiler detects these cases. When a scalar is used after the loop, but is not defined on every iteration of the loop, parallelization does not occur.

safeptr (nosafeptr)

The pragma safeptr directs the compiler to treat pointer variables of the indicated storage class as safe. The pragma nosafeptr directs the compiler to treat pointer variables of the indicated storage class as unsafe. This pragma alters the effects of the –Msafeptr command-line option.

The syntax of this pragma is:

```
cpgi$[] [no]safeptr={arg|local|auto|global|static|all},..
#pragma [scope] [no]safeptr={arg|local|auto|global|static|all},...
```

where scope is global, routine, or loop.

Note

The values local and auto are equivalent.

- all - All pointers are safe
- arg - Argument pointers are safe
- local - local pointers are safe
- global - global pointers are safe

- static - static local pointers are safe

In a file containing multiple functions, the command-line option –Msafeptr might be helpful for one function, but can't be used because another function in the file would produce incorrect results. In such a file, the safeptr pragma, used with routine scope could improve performance and produce correct results.

single (nosingle)

The pragma single directs the compiler not to implicitly convert float values to double non-prototyped functions. This can result in faster code if the program uses only float parameters.

Note

Since ANSI C specifies that floats must be converted to double, this pragma results in non-ANSI conforming code. Valid only for routine or global scope.

tp

You use the directive or pragma tp to specify one or more processor targets for which to generate code.

```
cpgi$ tp [target]...
```

Note

The tp directive or pragma can only be applied at the routine or global level. For more information about these levels, refer to"Scope of C/C++ Pragmas and Command-Line Options," on page 105.

Refer to the PGI Workstation Release Notes for a list of targets that can be used as parameters to the tp directive.

unroll (nounroll)

The unroll directive or pragma enables loop unrolling while nounroll disables loop unrolling.

Note

The unroll directive or pragma has no effect on vectorized loops.

The directive or pragma takes arguments c and n.

- c specifies that c complete unrolling should be turned on or off.
- n specifies that n (count) unrolling should be turned on or off. In addition, the following arguments may be added to the unroll directive:

In addition, the following arguments may be added to the unroll directive:

c:v sets the threshold to which c unrolling applies. v is a constant; and a loop whose constant loop count isless than or equal to (<=) v is completely unrolled.

```
cpgi$ unroll = c:v
```

n:v adjusts threshold to which n unrolling applies. v is a constant. A loop to which n unrolling applies is unrolled v times.

```
cpgi$ unroll = n:v
```

The directives unroll and nounroll only apply if `-Munroll` is selected on the command line.

vector (novector)

The directive or pragma novector disables vectorization. The directive or pragma vector re-enables vectorization after a previous novector directive. The directives vector and novector only apply if `-Mvect` has been selected on the command line.

vintr (novintr)

The directive or pragma novintr directs the vectorizer to disable recognition of vector intrinsics. The directive vintr is re-enables recognition of vector intrinsics after a previous novintr directive. The directives vintr and novintr only apply if `-Mvect` has been selected on the command line.

Prefetch Directives and Pragmas

As mentioned in Chapter 8, "*Using Directives and Pragmas*," on page 101, prefetch instructions can increase the speed of an application substantially by bringing data into cache so that it is available when the processor needs it. The PGI prefetch directive takes the form:

The syntax of a prefetch directive is as follows:

```
c$mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

The syntax of a prefetch pragma is as follows:

```
#pragma mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

For examples on how to use the prefetch directive or pragma, refer to "Prefetch Directives and Pragmas," on page 107.

!DEC$ Directives

As mentioned in Chapter 8, "*Using Directives and Pragmas*," on page 101, PGI Fortran compilers for Microsoft Windows support directives that help with inter-language calling and importing and exporting routines to and from DLLs. These directives all take the form:

```
!DEC$ directive
```

For specific format requirements, refer to "!DEC$ Directives," on page 109

ALIAS Directive

This directive specifies an alternative name with which to resolve a routine.

The syntax for the ALIAS directive is either of the following:

```
!DEC$ ALIAS routine_name , external_name
!DEC$ ALIAS routine_name : external_name
```

In this syntax, `external_name` is used as the external name for the specified `routine_name`.

If `external_name` is an identifier name, the name (in uppercase) is used as the external name for the specified `routine_name`. If `external_name` is a character constant, it is used as-is; the string is not changed to uppercase, nor are blanks removed.

You can also supply an alias for a routine using the ATTRIBUTES directive, described in the next section:

```
!DEC$ ATTIRIBUTES ALIAS : 'alias_name' :: routine_name
```

This directive specifies an alternative name with which to resolve a routine, as illustrated in the following code fragment that provides external names for three routines. In this fragment, the external name for `sub1` is `name1`, for `sub2` is `name2`, and for `sub3` is `name3`.

```
 subroutine sub
!DEC$ alias sub1 , 'name1'
!DEC$ alias sub2 : 'name2'
!DEC$ attributes alias : 'name3' :: sub3
```

ATTRIBUTES Directive

This directive lets you specify properties for data objects and procedures.

The syntax for the ATTRIBUTES directive is this:

```
!DEC$ ATTRIBUTES <list>
```

where <list> is one of the following:

ALIAS : 'alias_name' :: routine_name
: Specifies an alternative name with which to resolve `routine_name`.

C :: routine_name
: Specifies that the routine `routine_name` will have its arguments passed by value. When a routine marked C is called, arguments, except arrays, are sent by value. For characters, only the first character is passed. The standard Fortran calling convention is pass by reference.

DLLEXPORT :: name
: Specifies that `name` is being exported from a DLL.

DLLIMPORT :: name
: Specifies that `name` is being imported from a DLL.

NOMIXED_STR_LEN_ARG
: Specifies that hidden lengths are placed in sequential order at the end of the list, like `-Miface=unix`.

> ### Note
>
> This attribute only applies to routines that are CREF-style or that use the default Windows calling conventions.

REFERENCE :: name
: Specifies that the argument `name` is being passed by reference. Often this attribute is used in conjunction with `STDCALL`, where `STDCALL` refers to an entire routine; then individual arguments are modified with `REFERENCE`.

STDCALL :: routine_name
: Specifies that routine `routine_name` will have its arguments passed by value. When a routine marked `STDCALL` is called, arguments (except arrays and characters) will be sent by value. The standard Fortran calling convention is pass by reference.

VALUE :: name
: Specifies that the argument 'name' is being passed by value.

DECORATE Directive

The DECORATE directive specifies that the name specified in the ALIAS directive should have the prefix and postfix decorations performed on it that are associated with the calling conventions that are in effect. These declarations are the same ones performed on the name when ALIAS is not specified.

The syntax for the DECORATE directive is this:

```
!DEC$ DECORATE
```

Note

When ALIAS is not specified, this directive has no effect.

DISTRIBUTE Directive

This directive is front-end based, and tells the compiler at what point within a loop to split into two loops.

The syntax for the DISTRIBUTE directive is either of the following:

```
!DEC$ DISTRIBUTE POINT
```

```
!DEC$ DISTRIBUTEPOINT
```

Example:

```
 subroutine dist(a,b,n)
 integer i
 integer n
 integer a(*)
 integer b(*)
 do i = 1,n
 a(i) = a(i)+2
!DEC$ DISTRIBUTE POINT
 b(i) = b(i)*4
 enddo
 end subroutine
```

IGNORE_TKR Directive

This directive indicates to the compiler to ignore the type, kind, and/or rank of the specified dummy arguments in an interface of a procedure. The compiler also ignores the type, kind, and/or rank of the actual arguments when checking all the specifics in a generic call for ambiguities.

The syntax for the IGNORE_TKR directive is this:

```
!DIR$ IGNORE_TKR [ [(<letter>) <dummy_arg>] ... ]
```

<letter>
is one or any combination of the following:

T - type **K** - kind **R** - rank

For example, KR indicates to ignore both kind and rank rules and TKR indicates to ignore the type, kind, and rank arguments.

<dummy_arg>
if specified, indicates the dummy argument for which TKR rules should be ignored. If not specified, TKR rules are ignored for all dummy arguments in the procedure that contains the directive.

Rules

The following rules apply to this directive:

- IGNORE_TKR must not specify dummy arguments that are allocatable, Fortran 90 pointers, or assumed-shape arrays.
- IGNORE_TKR may only appear in the body of an interface block and may specify dummy argument names only.
- IGNORE_TKR may appear before or after the declarations of the dummy arguments it specifies.
- If dummy argument names are specified, IGNORE_TKR applies only to those particular dummy arguments.
- If no dummy argument names are specified, IGNORE_TKR applies to all dummy arguments except those that are allocatable objects, Fortran 90 pointers, or assumed-shape arrays.

Example:

Consider this subroutine fragment:

```
subroutine example(A,B,C,D)
!DIR$ IGNORE_TKR A, (R) B, (TK) C, (K) D
```

Table 20.1 indicates which rules are ignored for which dummy arguments in the sample subroutine fragment:

Table 20.1. IGNORE_TKR Example

Dummy Argument	Ignored Rules
A	Type, Kind and Rank
B	Only rank
C	Type and Kind
D	Only Kind

Notice that no letters were specified for A, so all type, kind, and rank rules are ignored.

Chapter 21. Run-time Environment

This chapter describes the programming model supported for compiler code generation, including register conventions and calling conventions for x86 and x64 processor-based systems. It addresses these conventions for processors running linux86 or Win32 operating systems, for processors running linux86-64 operating systems, and for processors running Win64 operating systems.

Note

In this chapter we sometimes refer to word, halfword, and double word. The equivalent byte information is word (4 byte), halfword (2 byte), and double word (8 byte).

Linux86 and Win32 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x86 processor running a linux86 or Win32 operating system. These standards must be followed to guarantee that compilers, application programs, and operating systems written by different people and organizations will work together. The conventions supported by the PGCC ANSI C compiler implement the application binary interface (ABI) as defined in the System V Application Binary Interface: Intel Processor Supplement and the System V Application Binary Interface, listed in the "Related Publications" section in the Preface.

Function Calling Sequence

This section describes the standard function calling sequence, including the stack frame, register usage, and parameter passing.

Register Usage Conventions

The following table defines the standard for register allocation. The 32-bit x86 Architecture provides a number of registers. All the integer registers and all the floating-point registers are global to all procedures in a running program.

Table 21.1. Register Allocation

Type	Name	Purpose
General	%eax	integer return value
	%edx	dividend register (for divide operations)
	%ecx	count register (shift and string operations)
	%ebx	local register variable
	%ebp	optional stack frame pointer
	%esi	local register variable
	%edi	local register variable
	%esp	stack pointer
Floating-point	%st(0)	floating-point stack top, return value
	%st(1)	floating-point next to stack top
	%st(...)	
	%st(7)	floating-point stack bottom

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. The next table shows the stack frame organization.

Table 21.2. Standard Stack Frame

Position	Contents	Frame
4n+8 (%ebp)	argument word n	previous
	argument words 1 to n-1	
8 (%ebp)	argument word 0	
4 (%ebp)	return address	
0 (%ebp)	caller's %ebp	current
-4 (%ebp)	n bytes of local	
-n (%ebp)	variables and temps	

Several key points concerning the stack frame:

- The stack is kept double word aligned.
- Argument words are pushed onto the stack in reverse order so the rightmost argument in C call syntax has the highest address. A dummy word may be pushed ahead of the rightmost argument in order to preserve doubleword alignment. All incoming arguments appear on the stack, residing in the stack frame of the caller.
- An argument's size is increased, if necessary, to make it a multiple of words. This may require tail padding, depending on the size of the argument.

All registers on an x86 system are global and thus visible to both a calling and a called function. Registers %ebp, %ebx, %edi, %esi, and %esp are non-volatile across function calls. Therefore, a function must preserve these registers' values for its caller. Remaining registers are volatile (scratch). If a calling function wants to preserve such a register value across a function call, it must save its value explicitly.

Some registers have assigned roles in the standard calling sequence:

%esp
: The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. At all times, the stack pointer should point to a word-aligned area.

%ebp
: The frame pointer holds a base address for the current stack frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from %ebp, while local variables reside in the current frame, referenced as negative offsets from %ebp. A function must preserve this register value for its caller.

%eax
: Integral and pointer return values appear in %eax. A function that returns a structure or union value places the address of the result in %eax. Otherwise, this is a scratch register.

%esi, %edi
: These local registers have no specified role in the standard calling sequence. Functions must preserve their values for the caller.

%ecx, %edx
: Scratch registers have no specified role in the standard calling sequence. Functions do not have to preserve their values for the caller.

%st(0)
: Floating-point return values appear on the top of the floating point register stack; there is no difference in the representation of single or double-precision values in floating point registers. If the function does not return a floating point value, then the stack must be empty.

%st(1) - %st(7)
: Floating point scratch registers have no specified role in the standard calling sequence. These registers must be empty before entry and upon exit from a function.

EFLAGS
: The flags register contains the system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" (i.e., zero) direction before entry and upon exit from a function. Other user flags have no specified role in the standard calling sequence and are not reserved.

Floating Point Control Word
: The control word contains the floating-point flags, such as the rounding mode and exception masking. This register is initialized at process initialization time and its value must be preserved.

Signals can interrupt processes. Functions called during signal handling have no unusual restriction on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without danger of signal handlers changing their values.

Function Return Values

Functions Returning No Value

Functions that return no value are also called procedures or void functions. These functions put no particular value in any register.

Functions Returning Scalars

- A function that returns an integral or pointer value places its result in register %eax.
- A function that returns a long long integer value places its result in the registers %edx and %eax. The most significant word is placed in %edx and the least significant word is placed in %eax.
- A floating-point return value appears on the top of the floating point stack. The caller must then remove the value from the floating point stack, even if it does not use the value. Failure of either side to meet its obligations leads to undefined program behavior. The standard calling sequence does not include any method to detect such failures nor to detect return value type mismatches. Therefore, the user must declare all functions properly. There is no difference in the representation of single-, double- or extended-precision values in floating-point registers.
- A call instruction pushes the address of the next instruction (the return address) onto the stack. The return instruction pops the address off the stack and effectively continues execution at the next instruction after the call instruction. A function that returns a scalar or no value must preserve the caller's registers. Additionally, the called function must remove the return address from the stack, leaving the stack pointer (%esp) with the value it had before the call instruction was executed.

Functions Returning Structures or Unions

If a function returns a structure or union, then the caller provides space for the return value and places its address on the stack as argument word zero. In effect, this address becomes a hidden first argument.

A function that returns a structure or union also sets %eax to the value of the original address of the caller's area before it returns. Thus, when the caller receives control again, the address of the returned object resides in register %eax and can be used to access the object. Both the calling and the called functions must cooperate to pass the return value successfully:

- The calling function must supply space for the return value and pass its address in the stack frame;
- The called function must use the address from the frame and copy the return value to the object so supplied;
- The called function must remove this address from the stack before returning.

Failure of either side to meet its obligation leads to undefined program behavior. The standard function calling sequence does not include any method to detect such failures nor to detect structure and union type mismatches. Therefore, you must declare the function properly.

The following table illustrates the stack contents when the function receives control, after the call instruction, and when the calling function again receives control, after the `ret` instruction.

Table 21.3. Stack Contents for Functions Returning struct/union

Position	After Call	After Return	Position
4n+8 (%esp)	argument word n	argument word n	4n-4 (%esp)
8 (%esp)	argument word 1	argument word 1	0 (%esp)
4 (%esp)	value address	undefined	
0 (%esp)	return address		

The following sections of this chapter describe where arguments appear on the stack. The examples in this chapter are written as if the function prologue is used.

Argument Passing

Integral and Pointer Arguments

As mentioned, a function receives all its arguments through the stack; the last argument is pushed first. In the standard calling sequence, the first argument is at offset 8(%ebp), the second argument is at offset 12(%ebp), as previously shown in Table 21.3, "Stack Contents for Functions Returning struct/union". Functions pass all integer-valued arguments as words, expanding or padding signed or unsigned bytes and halfwords as needed.

Table 21.4. Integral and Pointer Arguments

Call	Argument	Stack Address
g(1, 2, 3, (void *)0);	1	8 (%ebp)
	2	12 (%ebp)
	3	16 (%ebp)
	(void *) 0	20 (%ebp)

Floating-Point Arguments

The stack also holds floating-point arguments: single-precision values use one word and double-precision use two. The following example uses only double-precision arguments.

Table 21.5. Floating-point Arguments

Call	Argument	Stack Address
h(1.414, 1, 2.998e10);	word 0, 1.414	8 (%ebp)
	word 1, 1.414	12 (%ebp)
	1	16 (%ebp)
	word 0 2.998e10	20 (%ebp)
	word 1, 2.998e10	24 (%ebp)

Structure and Union Arguments

Structures and unions can have byte, halfword, or word alignment, depending on the constituents. An argument's size is increased, if necessary, to make it a multiple of words. This size increase may require tail

padding, depending on the size of the argument. Structure and union arguments are pushed onto the stack in the same manner as integral arguments. This process provides call-by-value semantics, letting the called function modify its arguments without affecting the calling function's object. In the following example, the argument, s, is a structure consisting of more than 2 words.

Table 21.6. Structure and Union Arguments

Call	Argument	Stack Address
i(1,s);	1	8 (%ebp)
	word 0, s	12 (%ebp)
	word 1, s	16 (%ebp)
	...	...

Implementing a Stack

In general, compilers and programmers must maintain a software stack. Register %esp is the stack pointer. Register %esp is set by the operating system for the application when the program is started. The stack must be a grow-down stack.

A separate frame pointer enables calls to routines that change the stack pointer to allocate space on the stack at run-time (e.g. alloca). Some languages can also return values from a routine allocated on stack space below the original top-of-stack pointer. Such a routine prevents the calling function from using %esp-relative addressing to get at values on the stack. If the compiler does not call routines that leave %esp in an altered state when they return, a frame pointer is not needed and is not used if the compiler option `-Mnoframe` is specified.

Although not required, the stack should be kept aligned on 8-byte boundaries so that 8-byte locals are favorably aligned with respect to performance. PGI's compilers allocate stack space for each routine in multiples of 8 bytes.

Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The `stdarg.h` and `varargs.h` files define several functions to access these parameters. A C routine with variable parameters must use the `va_start` macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters.

C Parameter Conversion

In C, for a called prototyped function, the parameter type in the called function must match the argument type in the calling function. If the called function is not prototyped, the calling convention uses the types of the arguments but promotes char or short to int, and unsigned char or unsigned short to unsigned int and promotes float to double, unless you use the `-Msingle` option. For more information on the `-Msingle` option, refer to Chapter 3. If the called function is prototyped, the unused bits of a register containing a char or short parameter are undefined and the called function must extend the sign of the unused bits when needed.

Example 21.1. C Program Calling an Assembly-language Routine

```
/* File: testmain.c */
main(){
 long l_para1 = 0x3f800000;
 float f_para2 = 1.0;
 double d_para3 = 0.5;
 float f_return;
 extern float sum_3 (long para1, float para2, double para3);
 f_return = sum_3(l_para1,f_para2, d_para3);
 printf("Parameter one, type long = %08x\n",l_para1);
 printf("Parameter two, type float = %f\n",f_para2);
 printf("Parameter three, type double = %g\n",d_para3);
 printf("The sum after conversion = %f\n",f_return);
}
```

```
# File: sum_3.s
# Computes ( para1 + para2 ) + para3
 .text
 .align 4
 .long .EN1-sum_3+0xc8000000
 .align 16
 .globl sum_3
sum_3:
 pushl %ebp
 movl %esp,%ebp
 subl $8,%esp
..EN1:
 fildl 8(%ebp)
 fadds 12(%ebp)
 faddl 16(%ebp)
 fstps -4(%ebp)
 flds -4(%ebp)
 addl $8,%esp
 leave
 ret
 .type sum_3,@function
 .size sum_3,.-sum_3
```

Linux86-64 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x64 processor running a linux86-64 operating system. These standards must be followed to guarantee that compilers, application programs, and operating systems written by different people and organizations will work together. The conventions supported by the PGCC ANSI C compiler implement the application binary interface (ABI) as defined in the System V Application Binary Interface: AMD64 Architecture Processor Supplement and the System V Application Binary Interface, listed in the "Related Publications" section in the Preface.

Note

The programming model used for Win64 and SUA64 differs from the Linux86-64 model. For more information, refer to "Win64 Programming Model," on page 356.

Function Calling Sequence

This section describes the standard function calling sequence, including the stack frame, register usage, and parameter passing.

Register Usage Conventions

The following table defines the standard for register allocation. The x64 Architecture provides a variety of registers. All the general purpose registers, XMM registers, and x87 registers are global to all procedures in a running program.

Table 21.7. Register Allocation

Type	Name	Purpose
General	%rax	1st return register
	%rbx	callee-saved; optional base pointer
	%rcx	pass 4th argument to functions
	%rdx	pass 3rd argument to functions; 2nd return register
	%rsp	stack pointer
	%rbp	callee-saved; optional stack frame pointer
	%rsi	pass 2nd argument to functions
	%rdi	pass 1st argument to functions
	%r8	pass 5th argument to functions
	%r9	pass 6th argument to functions
	%r10	temporary register; pass a function's static chain pointer
	%r11	temporary register
	%r12-r15	callee-saved registers
XMM	%xmm0-%xmm1	pass and return floating point arguments
	%xmm2-%xmm7	pass floating point arguments
	%xmm8-%xmm15	temporary registers
x87	%st(0)	temporary register; return long double arguments
	%st(1)	temporary register; return long double arguments
	%st(2) - %st(7)	temporary registers

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 21.8 shows the stack frame organization.

Table 21.8. Standard Stack Frame

Position	Contents	Frame
8n+16 (%rbp)	argument eightbyte n	previous

Position	Contents	Frame
	. . .	
16 (%rbp)	argument eightbyte 0	
8 (%rbp)	return address	current
0 (%rbp)	caller's %rbp	current
-8 (%rbp)	unspecified	
	. . .	
0 (%rsp)	variable size	
-128 (%rsp)	red zone	

Key points concerning the stack frame:

- The end of the input argument area is aligned on a 16-byte boundary.
- The 128-byte area beyond the location of %rsp is called the red zone and can be used for temporary local data storage. This area is not modified by signal or interrupt handlers.
- A call instruction pushes the address of the next instruction (the return address) onto the stack. The return instruction pops the address off the stack and effectively continues execution at the next instruction after the call instruction. A function must preserve non-volatile registers, a register whose contents must be preserved across subroutine calls. Additionally, the called function must remove the return address from the stack, leaving the stack pointer (%rsp) with the value it had before the call instruction was executed.

All registers on an x64 system are global and thus visible to both a calling and a called function. Registers %rbx, %rsp, %rbp, %r12, %r13, %r14, and %r15 are non-volatile across function calls. Therefore, a function must preserve these registers' values for its caller. Remaining registers are volatile (scratch) registers, that is a register whose contents need not be preserved across subroutine calls. If a calling function wants to preserve such a register value across a function call, it must save its value explicitly.

Registers are used extensively in the standard calling sequence. The first six integer and pointer arguments are passed in these registers (listed in order): %rdi, %rsi, %rdx, %rcx, %r8, %r9. The first eight floating point arguments are passed in the first eight XMM registers: %xmm0, %xmm1, …, %xmm7. The registers %rax and %rdx are used to return integer and pointer values. The registers %xmm0 and %xmm1 are used to return floating point values.

Additional registers with assigned roles in the standard calling sequence:

%rsp
: The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. The stack must be 16-byte aligned.

%rbp
: The frame pointer holds a base address for the current stack frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from %rbp, while local variables reside in the current frame, referenced as negative offsets from %rbp. A function must preserve this register value for its caller.

RFLAGS
: The flags register contains the system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" (i.e., zero) direction before entry and upon exit from a function. Other user flags have no specified role in the standard calling sequence and are not preserved.

Floating Point Control Word
: The control word contains the floating-point flags, such as the rounding mode and exception masking. This register is initialized at process initialization time and its value must be preserved.

Signals can interrupt processes. Functions called during signal handling have no unusual restriction on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without danger of signal handlers changing their values.

Function Return Values

Functions Returning Scalars or No Value

- A function that returns an integral or pointer value places its result in the next available register of the sequence %rax, %rdx.
- A function that returns a floating point value that fits in the XMM registers returns this value in the next available XMM register of the sequence %xmm0, %xmm1.
- An X87 floating-point return value appears on the top of the floating point stack in %st(0) as an 80-bit X87 number. If this X87 return value is a complex number, the real part of the value is returned in %st(0) and the imaginary part in %st(1).
- A function that returns a value in memory also returns the address of this memory in %rax.
- Functions that return no value (also called procedures or void functions) put no particular value in any register.

Functions Returning Structures or Unions

A function can use either registers or memory to return a structure or union. The size and type of the structure or union determine how it is returned. If a structure or union is larger than 16 bytes, it is returned in memory allocated by the caller.

To determine whether a 16-byte or smaller structure or union can be returned in one or more return registers, examine the first eight bytes of the structure or union. The type or types of the structure or union's fields making up these eight bytes determine how these eight bytes will be returned. If the eight bytes contain at least one integral type, the eight bytes will be returned in %rax even if non-integral types are also present in the eight bytes. If the eight bytes only contain floating point types, these eight bytes will be returned in %xmm0.

If the structure or union is larger than eight bytes but smaller than 17 bytes, examine the type or types of the fields making up the second eight bytes of the structure or union. If these eight bytes contain at least one integral type, these eight bytes will be returned in %rdx even if non-integral types are also present in the eight bytes. If the eight bytes only contain floating point types, these eight bytes will be returned in %xmm1.

If a structure or union is returned in memory, the caller provides the space for the return value and passes its address to the function as a "hidden" first argument in %rdi. This address will also be returned in %rax.

Argument Passing

Integral and Pointer Arguments

Integral and pointer arguments are passed to a function using the next available register of the sequence %rdi, %rsi, %rdx, %rcx, %r8, %r9. After this list of registers has been exhausted, all remaining integral and pointer arguments are passed to the function via the stack.

Floating-Point Arguments

Float and double arguments are passed to a function using the next available XMM register taken in the order from %xmm0 to %xmm7. After this list of registers has been exhausted, all remaining float and double arguments are passed to the function via the stack.

Structure and Union Arguments

Structure and union arguments can be passed to a function in either registers or on the stack. The size and type of the structure or union determine how it is passed. If a structure or union is larger than 16 bytes, it is passed to the function in memory.

To determine whether a 16-byte or smaller structure or union can be passed to a function in one or two registers, examine the first eight bytes of the structure or union. The type or types of the structure or union's fields making up these eight bytes determine how these eight bytes will be passed. If the eight bytes contain at least one integral type, the eight bytes will be passed in the first available general purpose register of the sequence %rdi, %rsi, %rdx, %rcx, %r8, %r9 even if non-integral types are also present in the eight bytes. If the eight bytes only contain floating point types, these eight bytes will be passed in the first available XMM register of the sequence from %xmm0 to %xmm7.

If the structure or union is larger than eight bytes but smaller than 17 bytes, examine the type or types of the fields making up the second eight bytes of the structure or union. If the eight bytes contain at least one integral type, the eight bytes will be passed in the next available general purpose register of the sequence %rdi, %rsi, %rdx, %rcx, %r8, %r9 even if non-integral types are also present in the eight bytes. If these eight bytes only contain floating point types, these eight bytes will be passed in the next available XMM register of the sequence from %xmm0 to %xmm7.

If the first or second eight bytes of the structure or union cannot be passed in a register for some reason, the entire structure or union must be passed in memory.

Passing Arguments on the Stack

If there are arguments left after every argument register has been allocated, the remaining arguments are passed to the function on the stack. The unassigned arguments are pushed on the stack in reverse order, with the last argument pushed first.

Table 21.9, "Register Allocation for Example A-2" shows the register allocation and stack frame offsets for the function declaration and call shown in the following example. Both table and example are adapted from System V Application Binary Interface: AMD64 Architecture Processor Supplement.

Example 21.2. Parameter Passing

```
typedef struct {
 int a, b;
 double d;
} structparam;
structparam s;
int e, f, g, h, i, j, k;
float flt;  double m, n;
extern void func(int e, int f, structparam s, int g, int h,
 float flt, double m, double n, int i, int j, int k);
void func2()
{
 func(e, f, s, g, h, flt, m, n, i, j, k);
}
```

Table 21.9. Register Allocation for Example A-2

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: e	%xmm0: s.d	0: j
%rsi: f	%xmm1: flt	8: k
%rdx: s.a,s.b	%xmm2: m	
%rcx: g	%xmm3: n	
%r8: h		
%r9: i		

Implementing a Stack

In general, compilers and programmers must maintain a software stack. The stack pointer, register %rsp, is set by the operating system for the application when the program is started. The stack must grow downwards from high addresses.

A separate frame pointer enables calls to routines that change the stack pointer to allocate space on the stack at run-time (e.g. alloca). Some languages can also return values from a routine allocated on stack space below the original top-of-stack pointer. Such a routine prevents the calling function from using %rsp-relative addressing for values on the stack. If the compiler does not call routines that leave %rsp in an altered state when they return, a frame pointer is not needed and may not be used if the compiler option –Mnoframe is specified.

The stack must be kept aligned on 16-byte boundaries.

Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The `stdarg.h` and `varargs.h` files define several functions to access these parameters. A C routine with variable parameters must use the va_start macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters.

For calls that use `varargs` or `stdargs`, the register %rax acts as a hidden argument whose value is the number of XMM registers used in the call.

C Parameter Conversion

In C, for a called prototyped function, the parameter type in the called function must match the argument type in the calling function. If the called function is not prototyped, the calling convention uses the types of the arguments but promotes char or short to int, and unsigned char or unsigned short to unsigned int and promotes float to double, unless you use the `–Msingle` option. For more information on the `–Msingle` option, refer to Chapter 3.

Calling Assembly Language Programs

Example 21.3. C Program Calling an Assembly-language Routine

```
/* File: testmain.c */
#include <stdio.h>
int
main() {
 long l_para1 = 2;
 float f_para2 = 1.0;
 double d_para3 = 0.5;
 float f_return;
 extern float sum_3(long para1, float para2, double para3);
 f_return = sum_3(l_para1, f_para2, d_para3);
 printf("Parameter one, type long = %ld\n", l_para1);
 printf("Parameter two, type float = %f\n", f_para2);
 printf("Parameter three, type double = %f\n", d_para3);
 printf("The sum after conversion = %f\n", f_return);
 return 0;
}
# File: sum_3.s
# Computes ( para1 + para2 ) + para3
 .text
 .align 16
 .globl sum_3
sum_3:
 pushq %rbp
 movq %rsp, %rbp
 cvtsi2ssq %rdi, %xmm2
 addss %xmm0, %xmm2
 cvtss2sd %xmm2,%xmm2
 addsd %xmm1, %xmm2
 cvtsd2ss %xmm2, %xmm2
 movaps %xmm2, %xmm0
 popq %rbp
 ret
 .type sum_3, @function
 .size sum_3,.-sum_3
```

Linux86-64 Fortran Supplement

Sections A2.4.1 through A2.4.4 of the ABI for x64 Linux and Mac OS X define the Fortran supplement. The register usage conventions set forth in that document remain the same for Fortran.

Fortran Fundamental Types

Table 21.10. Linux86-64 Fortran Fundamental Types

Fortran Type	Size (bytes)	Alignment (bytes)
INTEGER	4	4
INTEGER*1	1	1
INTEGER*2	2	2
INTEGER*4	4	4
INTEGER*8	8	8
LOGICAL	4	4
LOGICAL*1	1	1
LOGICAL*2	2	2
LOGICAL*4	4	4
LOGICAL*8	8	8
BYTE	1	1
CHARACTER*n	n	1
REAL	4	4
REAL*4	4	4
REAL*8	8	8
DOUBLE PRECISION	8	8
COMPLEX	8	4
COMPLEX*8	8	4
COMPLEX*16	16	8
DOUBLE COMPLEX	16	8

A logical constant is one of:

- .TRUE.
- .FALSE.

The logical constants .TRUE. and .FALSE. are defined to be the four-byte values -1 and 0 respectively. A logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise.

Note that the value of a character is not automatically NULL-terminated.

Naming Conventions

By default, all globally visible Fortran symbol names (subroutines, functions, common blocks) are converted to lower-case. In addition, an underscore is appended to Fortran global names to distinguish the Fortran name space from the C/C++ name space.

Argument Passing and Return Conventions

Arguments are passed by reference (i.e. the address of the argument is passed, rather than the argument itself). In contrast, C/C++ arguments are passed by value.

When passing an argument declared as Fortran type CHARACTER, an argument representing the length of the CHARACTER argument is also passed to the function. This length argument is a four-byte integer passed by value, and is passed at the end of the parameter list following the other formal arguments. A length argument is passed for each CHARACTER argument; the length arguments are passed in the same order as their respective CHARACTER arguments.

A Fortran function, returning a value of type CHARACTER, adds two arguments to the beginning of its argument list. The first additional argument is the address of the area created by the caller for the return value; the second additional argument is the length of the return value. If a Fortran function is declared to return a character value of constant length, for example CHARACTER*4 FUNCTION CHF(), the second extra parameter representing the length of the return value must still be supplied.

A Fortran complex function returns its value in memory. The caller provides space for the return value and passes the address of this storage as if it were the first argument to the function.

Alternate return specifiers of a Fortran function are not passed as arguments by the caller. The alternate return function passes the appropriate return value back to the caller in %rax.

The handling of the following Fortran 90 features is implementation-defined: internal procedures, pointer arguments, assumed-shape arguments, functions returning arrays, and functions returning derived types.

Inter-language Calling

Inter-language calling between Fortran and C/C++ is possible if function/subroutine parameters and return values match types. If a C/C++ function returns a value, call it from Fortran as a function, otherwise, call it as a subroutine. If a Fortran function has type CHARACTER or COMPLEX, call it from C/C++ as a void function. If a Fortran subroutine has alternate returns, call it from C/C++ as a function returning int; the value of such a subroutine is the value of the integer expression specified in the alternate RETURN statement. If a Fortran subroutine does not contain alternate returns, call it from C/C++ as a void function.

Table 21.11 provides the C/C++ data type corresponding to each Fortran data type.

Table 21.11. Fortran and C/C++ Data Type Compatibility

Fortran Type	C/C++ Type	Size (bytes)
CHARACTER*n x	char x[n]	n
REAL x	float x	4
REAL*4 x	float x	4
REAL*8 x	double x	8
DOUBLE PRECISION x	double x	8
INTEGER x	int x	4
INTEGER*1 x	signed char x	1

Fortran Type	C/C++ Type	Size (bytes)
INTEGER*2 x	short x	2
INTEGER*4 x	int x	4
INTEGER*8 x	long x, or long long x	8
LOGICAL x	int x	4
LOGICAL*1 x	char x	1
LOGICAL*2 x	short x	2
LOGICAL*4 x	int x	4
LOGICAL*8 x	long x, or long long x	8

Table 21.12. Fortran and C/C++ Representation of the COMPLEX Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8
complex*8 x	struct {float r,i;} x; float complex x;	8 8
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16

Note

For C/C++, the `complex` type implies C99 or later.

Arrays

C/C++ arrays and Fortran arrays use different default initial array index values. By default, C/C++ arrays start at 0 and Fortran arrays start at 1. A Fortran array can be declared to start at zero.

Another difference between Fortran and C/C++ arrays is the storage method used. Fortran uses column-major order and C/C++ use row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed. Inter-language function mixing is not recommended for arrays other than single dimensional arrays and square two-dimensional arrays.

Structures, Unions, Maps, and Derived Types

Fields within Fortran structures and derived types, and multiple map declarations within a Fortran union, conform to the same alignment requirements used by C structures.

Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore.

For example, the Fortran common block:

```
INTEGER I, J
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, j, c, cd, d
```

is represented in C with the following equivalent:

```
extern struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

and in C++ with the following equivalent:

```
extern "C" struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

Note

The compiler-provided name of the BLANK COMMON block is implementation specific.

Calling Fortran COMPLEX and CHARACTER functions from C/C++ is not as straightforward as calling other types of Fortran functions. Additional arguments must be passed to the Fortran function by the C/C++ caller. A Fortran COMPLEX function returns its value in memory; the first argument passed to the function must contain the address of the storage for this value. A Fortran CHARACTER function adds two arguments to the beginning of its argument list. The following example of calling a Fortran CHARACTER function from C/C++ illustrates these caller-provided extra parameters:

```
CHARACTER*(*) FUNCTION CHF(C1, I)
CHARACTER*(*) C1
INTEGER I
END
```

```
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

The extra parameters tmp and 10 are supplied for the return value, while 9 is supplied as the length of c1. Refer to Section 2.8, Argument Passing and Return Conventions, for additional information.

Win64 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x64 processor running a Win64 operating system, including SUA64. These standards must be followed to guarantee that compilers, application programs, and operating systems written by different people and organizations will work together. The conventions supported by the PGCC ANSI C compiler implement the application binary interface (ABI) as defined in the AMD64 Software Conventions document.

Function Calling Sequence

This section describes the standard function calling sequence, including the stack frame, register usage, and parameter passing.

Register Usage Conventions

The following table defines the standard for register allocation. The 64-bit AMD64 and Intel 64 architectures provide a number of registers. All the general purpose registers, XMM registers, and x87 registers are global to all procedures in a running program.

Table 21.13. Register Allocation

Type	Name	Purpose
General	%rax	return value register
	%rbx	callee-saved
	%rcx	pass 1st argument to functions
	%rdx	pass 2nd argument to functions
	%rsp	stack pointer
	%rbp	callee-saved; optional stack frame pointer
	%rsi	callee-saved
	%rdi	callee-saved
	%r8	pass 3rd argument to functions
	%r9	pass 4th argument to functions
	%r10-%r11	temporary registers; used in syscall/sysret instructions
	%r12-r15	callee-saved registers
XMM	%xmm0	pass 1st floating point argument; return value register
	%xmm1	pass 2nd floating point argument
	%xmm2	pass 3rd floating point argument
	%xmm3	pass 4th floating point argument
	%xmm4-%xmm5	temporary registers
	%xmm6-%xmm15	callee-saved registers

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 21.14 shows the stack frame organization.

Table 21.14. Standard Stack Frame

Position	Contents	Frame
8n-120 (%rbp)	argument eightbyte n	previous
	. . .	
-80 (%rbp)	argument eightbyte 5	
-88 (%rbp)	%r9 home	
-96 (%rbp)	%r8 home	
-104 (%rbp)	%rdx home	
-112 (%rbp)	%rcx home	
-120 (%rbp)	return address	current
-128 (%rbp)	caller's %rbp	
	. . .	
0 (%rsp)	variable size	

Key points concerning the stack frame:

- The parameter area at the bottom of the stack must contain enough space to hold all the parameters needed by any function call. Space must be set aside for the four register parameters to be "homed" to the stack even if there are less than four register parameters used in a given call.
- Sixteen-byte alignment of the stack is required except within a function's prolog and within leaf functions.

All registers on an x64 system are global and thus visible to both a calling and a called function. Registers %rbx, %rsp, %rbp, %rsi, %rdi, %r12, %r13, %r14, and %r15 are non-volatile. Therefore, a called function must preserve these registers' values for its caller. Remaining registers are scratch. If a calling function wants to preserve such a register value across a function call, it must save a value in its local stack frame.

Registers are used in the standard calling sequence. The first four arguments are passed in registers. Integral and pointer arguments are passed in these general purpose registers (listed in order): %rcx, %rdx, %r8, %r9. Floating point arguments are passed in the first four XMM registers: %xmm0, %xmm1, %xmm2, %xmm3. Registers are assigned using the argument's ordinal position in the argument list. For example, if a function's first argument is an integral type and its second argument is a floating-point type, the first argument will be passed in the first general purpose register (%rcx) and the second argument will be passed in the second XMM register (%xmm1); the first XMM register and second general purpose register are ignored. Arguments after the first four are passed on the stack.

Integral and pointer type return values are returned in %rax. Floating point return values are returned in %xmm0.

Additional registers with assigned roles in the standard calling sequence:

%rsp
: The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. The stack pointer should point to a 16-byte aligned area unless in the prolog or a leaf function.

%rbp
: The frame pointer, if used, can provide a way to reference the previous frames on the stack. Details are implementation dependent. A function must preserve this register value for its caller.

MXCSR
: The flags register MXCSR contains the system flags, such as the direction flag and the carry flag. The six status flags (MXCSR[0:5]) are volatile; the remainder of the register is nonvolatile.

x87 - Floating Point Control Word (FPCSR)
: The control word contains the floating-point flags, such as the rounding mode and exception masking. This register is initialized at process initialization time and its value must be preserved.

Signals can interrupt processes. Functions called during signal handling have no unusual restriction on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without danger of signal handlers changing their values.

Function Return Values

Functions Returning Scalars or No Value

- A function that returns an integral or pointer value that fits in 64 bits places its result in %rax.
- A function that returns a floating point value that fits in the XMM registers returns this value in %xmm0.
- A function that returns a value in memory via the stack places the address of this memory (passed to the function as a "hidden" first argument in %rcx) in %rax.
- Functions that return no value (also called procedures or void functions) put no particular value in any register.
- A call instruction pushes the address of the next instruction (the return address) onto the stack. The return instruction pops the address off the stack and effectively continues execution at the next instruction after the call instruction. A function that returns a scalar or no value must preserve the caller's registers as previously described. Further, the called function must remove the return address from the stack, leaving the stack pointer (%rsp) with the value it had before the call instruction was executed.

Functions Returning Structures or Unions

A function can use either registers or the stack to return a structure or union. The size and type of the structure or union determine how it is returned. A structure or union is returned in memory if it is larger than 8 bytes or if its size is 3, 5, 6, or 7 bytes. A structure or union is returned in %rax if its size is 1, 2, 4, or 8 bytes.

If a structure or union is to be returned in memory, the caller provides space for the return value and passes its address to the function as a "hidden" first argument in %rcx. This address will also be returned in %rax.

Argument Passing

Integral and Pointer Arguments

Integral and pointer arguments are passed to a function using the next available register of the sequence %rcx, %rdx, %r8, %r9. After this list of registers has been exhausted, all remaining integral and pointer arguments are passed to the function via the stack.

Floating-Point Arguments

Float and double arguments are passed to a function using the next available XMM register of the sequence %xmm0, %xmm1, %xmm2, %xmm3. After this list of registers has been exhausted, all remaining XMM floating-point arguments are passed to the function via the stack.

Array, Structure, and Union Arguments

Arrays and strings are passed to functions using a pointer to caller-allocated memory.

Structure and union arguments of size 1, 2, 4, or 8 bytes will be passed as if they were integers of the same size. Structures and unions of other sizes will be passed as a pointer to a temporary, allocated by the caller, and whose value contains the value of the argument. The caller-allocated temporary memory used for arguments of aggregate type must be 16-byte aligned.

Passing Arguments on the Stack

Registers are assigned using the argument's ordinal position in the argument list. For example, if a function's first argument is an integral type and its second argument is a floating-point type, the first argument will be passed in the first general purpose register (%rcx) and the second argument will be passed in the second XMM register (%xmm1); the first XMM register and second general purpose register are ignored. Arguments after the first four are passed on the stack; they are pushed on the stack in reverse order, with the last argument pushed first.

Table 21.15, "Register Allocation for Example A-4" shows the register allocation and stack frame offsets for the function declaration and call shown in the following example.

Example 21.4. Parameter Passing

```
typedef struct {
 int i;
 float f;
} struct1;
int i;
float f;
double d;
long l;
long long ll;
struct1 s1;
extern void func (int i, float f, struct1 s1, double d, long long ll, long l);
func (i, f, s1, d, ll, l);
```

Table 21.15. Register Allocation for Example A-4

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rcx: i	%xmm0: <ignored>	32: ll
%rdx: <ignored>	%xmm1: f	40: l
%r8: s1.i, s1.f	%xmm2: <ignored>	
%r9: <ignored>	%xmm3: d	

Implementing a Stack

In general, compilers and programmers must maintain a software stack. The stack pointer, register %rsp, is set by the operating system for the application when the program is started. The stack must grow downwards from high addresses.

A separate frame pointer enables calls to routines that change the stack pointer to allocate space on the stack at run-time (e.g. alloca). Some languages can also return values from a routine allocated on stack space below the original top-of-stack pointer. Such a routine prevents the calling function from using %rsp-relative addressing to get at values on the stack. If the compiler does not call routines that leave %rsp in an altered state when they return, a frame pointer is not needed and is not used if the compiler option `-Mnoframe` is specified.

The stack must always be 16-byte aligned except within the prolog and within leaf functions.

Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The `stdarg.h` and `varargs.h` files define several functions to access these parameters. A C routine with variable parameters must use the `va_start` macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters.

For unprototyped functions or functions that use varargs, floating-point arguments passed in registers must be passed in both an XMM register and its corresponding general purpose register.

C Parameter Conversion

In C, for a called prototyped function, the parameter type in the called function must match the argument type in the calling function. If the called function is not prototyped, the calling convention uses the types of the arguments but promotes char or short to int, and unsigned char or unsigned short to unsigned int and promotes float to double, unless you use the `-Msingle` option. For more information on the `-Msingle` option, refer to Chapter 3. If the called function is prototyped, the unused bits of a register containing a char or short parameter are undefined and the called function must extend the sign of the unused bits when needed.

Calling Assembly Language Programs

Example 21.5. C Program Calling an Assembly-language Routine

```
/* File: testmain.c */
```

```
main() {
 long l_para1 = 0x3f800000;
 float f_para2 = 1.0;
 double d_para3 = 0.5;
 float f_return;
 extern float sum_3 (long para1, float para2, double para3);
 f_return = sum_3(l_para1,f_para2, d_para3);
 printf("Parameter one, type long = %08x\n",l_para1);
 printf("Parameter two, type float = %f\n",f_para2);
 printf("Parameter three, type double = %g\n",d_para3);
 printf("The sum after conversion = %f\n",f_return);
}
# File: sum_3.s
# Computes ( para1 + para2 ) + para3
 .text
 .align 16
 .globl sum_3
sum_3:
 pushq %rbp
 leaq 128(%rsp), %rbp
 cvtsi2ss %ecx, %xmm0
 addss %xmm1, %xmm0
 cvtss2sd %xmm0, %xmm0
 addsd %xmm2, %xmm0
 cvtsd2ss %xmm0, %xmm0
 popq %rbp
 ret
 .type sum_3,@function
 .size sum_3,.-sum_3
```

Win64/SUA64 Fortran Supplement

Sections A3.4.1 through A3.4.4 of the AMD64 Software Conventions for Win64 define the Fortran supplement. The register usage conventions set forth in that document remain the same for Fortran.

Fortran Fundamental Types

Table 21.16. Win64 Fortran Fundamental Types

Fortran Type	Size (bytes)	Alignment (bytes)
INTEGER	4	4
INTEGER*1	1	1
INTEGER*2	2	2
INTEGER*4	4	4
INTEGER*8	8	8
LOGICAL	4	4
LOGICAL*1	1	1
LOGICAL*2	2	2
LOGICAL*4	4	4

Fortran Type	Size (bytes)	Alignment (bytes)
LOGICAL*8	8	8
BYTE	1	1
CHARACTER*n	n	1
REAL	4	4
REAL*4	4	4
REAL*8	8	8
DOUBLE PRECISION	8	8
COMPLEX	8	4
COMPLEX*8	8	4
COMPLEX*16	16	8
DOUBLE COMPLEX	16	8

A logical constant is one of:

- .TRUE.
- .FALSE.

The logical constants .TRUE. and .FALSE. are defined to be the four-byte value 1 and 0 respectively. A logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise.

Note that the value of a character is not automatically NULL-terminated.

Fortran Naming Conventions

By default, all globally visible Fortran symbol names (subroutines, functions, common blocks) are converted to lower-case. In addition, an underscore is appended to Fortran global names to distinguish the Fortran name space from the C/C++ name space.

Fortran Argument Passing and Return Conventions

Arguments are passed by reference, meaning the address of the argument is passed rather than the argument itself. In contrast, C/C++ arguments are passed by value.

When passing an argument declared as Fortran type CHARACTER, an argument representing the length of the CHARACTER argument is also passed to the function. This length argument is a four-byte integer passed by value, and is passed at the end of the parameter list following the other formal arguments. A length argument is passed for each CHARACTER argument; the length arguments are passed in the same order as their respective CHARACTER arguments.

A Fortran function, returning a value of type CHARACTER, adds two arguments to the beginning of its argument list. The first additional argument is the address of the area created by the caller for the return value; the second additional argument is the length of the return value. If a Fortran function is declared to return a

character value of constant length, for example `CHARACTER*4 FUNCTION CHF()`, the second extra parameter representing the length of the return value must still be supplied.

A Fortran complex function returns its value in memory. The caller provides space for the return value and passes the address of this storage as if it were the first argument to the function.

Alternate return specifiers of a Fortran function are not passed as arguments by the caller. The alternate return function passes the appropriate return value back to the caller in %rax.

The handling of the following Fortran 90 features is implementation-defined: internal procedures, pointer arguments, assumed-shape arguments, functions returning arrays, and functions returning derived types.

Inter-language Calling

Inter-language calling between Fortran and C/C++ is possible if function/subroutine parameters and return values match types. If a C/C++ function returns a value, call it from Fortran as a function, otherwise, call it as a subroutine. If a Fortran function has type CHARACTER or COMPLEX, call it from C/C++ as a void function. If a Fortran subroutine has alternate returns, call it from C/C++ as a function returning int; the value of such a subroutine is the value of the integer expression specified in the alternate RETURN statement. If a Fortran subroutine does not contain alternate returns, call it from C/C++ as a void function.

Table 21.17 provides the C/C++ data type corresponding to each Fortran data type.

Table 21.17. Fortran and C/C++ Data Type Compatibility

Fortran Type	C/C++ Type	Size (bytes)
CHARACTER*n x	char x[n]	n
REAL x	float x	4
REAL*4 x	float x	4
REAL*8 x	double x	8
DOUBLE PRECISION x	double x	8
INTEGER x	int x	4
INTEGER*1 x	signed char x	1
INTEGER*2 x	short x	2
INTEGER*4 x	int x	4
INTEGER*8 x	long long x	8
LOGICAL x	int x	4
LOGICAL*1 x	char x	1
LOGICAL*2 x	short x	2
LOGICAL*4 x	int x	4
LOGICAL*8 x	long long x	8

Table 21.18 provides the Fortran and C/C++ representation of the COMPLEX type.

Table 21.18. Fortran and C/C++ Representation of the COMPLEX Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8 8
complex*8 x	struct {float r,i;} x; float complex x;	8 8
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16

Note

For C/C++, the `complex` type implies C99 or later.

Arrays

C/C++ arrays and Fortran arrays use different default initial array index values. By default, C/C++ arrays start at 0 and Fortran arrays start at 1. A Fortran array can be declared to start at zero.

Another difference between Fortran and C/C++ arrays is the storage method used. Fortran uses column-major order and C/C++ use row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed. Inter-language function mixing is not recommended for arrays other than single dimensional arrays and square two-dimensional arrays.

Structures, Unions, Maps, and Derived Types.

Fields within Fortran structures and derived types, and multiple map declarations within a Fortran union, conform to the same alignment requirements used by C structures.

Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore. For example, the Fortran common block:

```
INTEGER I, J
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, j, c, cd, d
```

is represented in C with the following equivalent:

```
extern struct {
int i;
int j;
```

```
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

and in C++ with the following equivalent:

```
extern "C" struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

Note

The compiler-provided name of the BLANK COMMON block is implementation-specific.

Calling Fortran COMPLEX and CHARACTER functions from C/C++ is not as straightforward as calling other types of Fortran functions. Additional arguments must be passed to the Fortran function by the C/C++ caller. A Fortran COMPLEX function returns its value in memory; the first argument passed to the function must contain the address of the storage for this value. A Fortran CHARACTER function adds two arguments to the beginning of its argument list. The following example of calling a Fortran CHARACTER function from C/C++ illustrates these caller-provided extra parameters:

```
CHARACTER*(*) FUNCTION CHF(C1, I)
CHARACTER*(*) C1
INTEGER I
END
```

```
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

The extra parameters `tmp` and `10` are supplied for the return value, while `9` is supplied as the length of `c1`. Refer to "Argument Passing and Return Values," on page 151, for additional information.

Chapter 22. C++ Dialect Supported

The PGC++ compiler accepts the C++ language of the ISO/IEC 14882:1998 C++ standard, except for Exported Templates.PGC++ optionally accepts a number of features erroneously accepted by cfront version 2.1 or 3.0. Using the -b option, PGC++ accepts these features, which may never have been legal C++, but have found their way into some user's code.

Command-line options provide full support of many C++ variants, including strict standard conformance. PGC++ provides command line options that enable the user to specify whether anachronisms and/or cfront 2.1/3.0 compatibility features should be accepted.

Extensions Accepted in Normal C++ Mode

The following extensions are accepted in all modes, except when strict ANSI violations are diagnosed as errors, described in the –A option:

- A friend declaration for a class may omit the class keyword:

```
class A {
friend B; // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes:

```
class A {
const int size = 10;
int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A{
int A::f(); // Should be int f();
}
```

- The preprocessing symbol c_plusplus is defined in addition to the standard __cplusplus.
- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a "default'' assignment operator --- that is, such a declaration blocks the implicit generation of

a copy assignment operator. (This is cfront behavior that is known to be relied upon in at least one widely used library.) Here's an example:

```
struct A { } ;
struct B : public A {
B& operator=(A&);
};
```

- By default, as well as in cfront-compatibility mode, there will be no implicit declaration of B::operator=(const B&), whereas in strict-ANSI mode B::operator=(A&) is not a copy assignment operator and B::operator=(const B&) is implicitly declared.
- Implicit type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted. Here's an example:

```
extern "C" void
f(); // f's type has extern "C" linkage
 void (*pf) () // pf points to an extern
"C++" function
 = &f; // error unless
implicit conv is allowed
```

cfront 2.1 Compatibility Mode

The following extensions are accepted in cfront 2.1 compatibility mode in addition to the extensions listed in the following section. These things were corrected in the 3.0 release of cfront:

- The dependent statement of an if, while, do-while, or for is not considered to define a scope. The dependent statement may not be a declaration. Any objects constructed within the dependent statement are destroyed at exit from the dependent statement.
- Implicit conversion from integral types to enumeration types is allowed.
- A non-const member function may be called for a const object. A warning is issued.
- A const void * value may be implicitly converted to a void * value, e.g., when passed as an argument.
- When, in determining the level of argument match for overloading, a reference parameter is initialized from an argument that requires a non-class standard conversion, the conversion counts as a user-defined conversion. (This is an outright bug, which unfortunately happens to be exploited in some class libraries.)
- When a builtin operator is considered alongside overloaded operators in overload resolution, the match of an operand of a builtin type against the builtin type required by the builtin operator is considered a standard conversion in all cases (e.g., even when the type is exactly right without conversion).
- A reference to a non-const type may be initialized from a value that is a const-qualified version of the same type, but only if the value is the result of selecting a member from a const class object or a pointer to such an object.
- A cast to an array type is allowed; it is treated like a cast to a pointer to the array element type. A warning is issued.
- When an array is selected from a class, the type qualifiers on the class object (if any) are not preserved in the selected array. (In the normal mode, any type qualifiers on the object are preserved in the element type of the resultant array.)

- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.
- An expression of type void may be supplied on the return statement in a function with a void return type. A warning is issued.
- cfront has a bug that causes a global identifier to be found when a member of a class or one of its base classes should actually be found. This bug is not emulated in cfront compatibility mode.
- A parameter of type "const void *'' is allowed on operator delete; it is treated as equivalent to "void *".
- A period (".") may be used for qualification where "::" should be used. Only "::'' may be used as a global qualifier. Except for the global qualifier, the two kinds of qualifier operators may not be mixed in a given name (i.e., you may say A::B::C or A.B.C but not A::B.C or A.B::C). A period may not be used in a vacuous destructor reference nor in a qualifier that follows a template reference such as A<T>::B.
- cfront 2.1 does not correctly look up names in friend functions that are inside class definitions. In this example function f should refer to the functions and variables (e.g., f1 and a1) from the class declaration. Instead, the global definitions are used.

```
int a1;
int e1;
void f1();
class A {
 int a1;
 void f1();
 friend void f()
 {
 int i1 = a1; // cfront uses global a1
 f1(); // cfront uses global f1
 }
};
```

- Only the innermost class scope is (incorrectly) skipped by cfront as illustrated in the following example.

```
int a1;
int b1;
struct A {
 static int a1;
 class B {
 static int b1;
 friend void f()
 {
 int i1 = a1; // cfront uses A::a1
 int j1 = b1; // cfront uses global b1
 }
 };
};
```

- operator= may be declared as a nonmember function. (This is flagged as an anachronism by cfront 2.1)
- A type qualifier is allowed (but ignored) on the declaration of a constructor or destructor. For example:

```
class A {
 A() const; // No error in cfront 2.1 mode
};
```

cfront 2.1/3.0 Compatibility Mode

The following extensions are accepted in both cfront 2.1 and cfront 3.0 compatibility mode (i.e., these are features or problems that exist in both cfront 2.1 and 3.0):

- Type qualifiers on this parameter may to be dropped in contexts such as this example:

```
struct
A {
 void f() const;
};
void (A::*fp)() = &A::f;
```

  This is actually a safe operation. A pointer to a const function may be put into a pointer to non-const, because a call using the pointer is permitted to modify the object and the function pointed to will actually not modify the object. The opposite assignment would not be safe.

- Conversion operators specifying conversion to void are allowed.
- A nonstandard friend declaration may introduce a new type. A friend declaration that omits the elaborated type specifier is allowed in default mode, but in cfront mode the declaration is also allowed to introduce a new type name.

```
 struct A {
 friend B;
 };
```

- The third operator of the ? operator is a conditional expression instead of an assignment expression.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example,

```
 int *p;
 const int *&r = p; // No
temporary used
```

- A reference may be initialized with a null.

Chapter 23. Fortran Module/Library Interfaces for Windows

PGI Fortran for Windows provides access to a number of libraries that export C interfaces by using Fortran modules. PGI uses this mechanism to support the Win32 API and Unix/Linux/Mac OS X portability libraries. This chapter describes the Fortran module library interfaces that PGI supports, describing each property available.

Source Files

All routines described in this chapter have their prototypes and interfaces described in source files that are included in the PGI Windows compiler installation. The location of these files depends on your operating system version, either win32 or win64, and the release version that you have installed, such as 7.2-5 or 9.0-0. These files are typically located in this directory:

```
C:/Program Files/PGI/{win32,win64}/[release_version]/src
```

For example, if you have installed the Win32 version of the 9.0-0 release, look for your files in this location:

```
C:/Program Files/PGI/win32/9.0-0/src
```

Data Types

Because the Win32 API and Portability interfaces resolve to C language libraries, it is important to understand how the data types compare within the two languages. Here is a table summarizing how C types correspond with Fortran types for some of the more common data types:

Table 23.1. Fortran Data Type Mappings

Windows Data Type	Fortran Data Type
BOOL	LOGICAL(4)
BYTE	BYTE
CHAR	CHARACTER
SHORT, WORD	INTEGER(2)

Windows Data Type	Fortran Data Type
DWORD, INT, LONG	INTEGER(4)
LONG LONG	INTEGER(8)
FLOAT	REAL(4)
DOUBLE	REAL(8)
x86 Pointers	INTEGER(4)
x64 Pointers	INTEGER(8)

For more information on data types, refer to "Fortran Data Types," on page 187.

Using DFLIB and DFPORT

PGI includes Fortran module interfaces to libraries supporting some standard C library and Unix/Linux/Mac OS X system call functionality. These functions are provided by the `DFLIB` and `DFPORT` modules. To utilize these modules, add the appropriate `USE` statement:

```
use dflib
```

```
use dfport
```

DFLIB

The following table lists the functions that `DFLIB` includes. In the table `[Generic]` refers to a generic routine. To view the prototype and interfaces, look in the location described in "Source Files," on page 371.

Table 23.2. DFLIB Function Summary

Routine	Result	Description
commitqq	LOGICAL*4	Executes any pending write operations for the file associated with the specified unit to the file's physical device.
delfilesqq	INTEGER*4	Deletes the specified files in a specified directory.
findfileqq	INTEGER*4	Searches for a file in the directories specified in the PATH environment variable.
fullpathqq	INTEGER*4	Returns the full path for a specified file or directory.
getdat	INTEGER*2,*4,*8	[Generic] Returns the date.
getdrivedirqq	INTEGER*4	Returns the current drive and directory path.
getenvqq	INTEGER*4	Returns a value from the current environment.
getfileinfoqq	INTEGER*4	Returns information about files with names that match the specified string.
getfileinfoqqi8	INTEGER*4	Returns information about files with names that match the specified string.
gettim	INTEGER*2,*4,*8	[Generic] Returns the time.
packtimeqq	INTEGER*4	Packs the time and date values for use by setfiletimeqq
renamefileqq	LOGICAL*4	Renames the specified file.

Routine	Result	Description
runqq	INTEGER*2	Calls another program and waits for it to execute.
setenvqq	LOGICAL*4	Sets the values of an existing environment variable or adds a new one.
setfileaccessqq	LOGICAL*4	Sets the file access mode for the specified file.
setfiletimeqq	LOGICAL*4	Sets the modification time for the specified file.
signalqq	INTEGER*8	Controls signal handling.
sleepqq	None	Delays execution of the program for a specified time.
splitpathqq	LOGICAL*4	Breaks a full path into components.
systemqq	LOGICAL*4	Executes a command by passing a command string to the operating system's command interpreter.
unpacktimeqq	Multiple INTEGERS	Unpacks a file's packed time and date value into its component parts.

DFPORT

The following table lists the functions that `DFPORT` includes. In the table `[Generic]` refers to a generic routine. To view the prototype and interfaces, look in the location described in "Source Files," on page 371.

Table 23.3. DFPORT Functions

Routine	Result	Description
abort	None	Immediately terminates the program. If the operating systems supports a core dump, abort produces one that can be used for debugging.
access	INTEGER*4	Determines access mode or existence of a file.
alarm	INTEGER*4	Executes a routine after a specified time.
besj0	REAL*4	Computes the BESSEL function of the first kind of order 0 of X, where X is real.
besj1	REAL*4	Computes the BESSEL function of the first kind of order 1 of X, where X is real.
besjn	REAL*4	Computes the BESSEL function of the first kind of order N of X, where N is an integer and X is real.
besy0	REAL*4	Computes the BESSEL function of the second kind of order 0 of X, where X is real.
besy1	REAL*4	Computes the BESSEL function of the second kind of order 1 of X, where X is real.
besyn	REAL*4	Computes the BESSEL function of the second kind of order N of X, where N is an integer and X is real.
chdir	INTEGER*4	Changes the current directory to the directory specified. Returns 0, if successful or an error

Routine	Result	Description
chmod	INTEGER*4	Changes the mode of a file by setting the access permissions of the specified file to the specified mode. Returns 0 if successful, or error
ctime	STRING(24)	Converts and returns the specified time and date as a string.
date	STRING	Returns the date as a character string: dd-mm-yy.
dbesj0	REAL*8	Computes the double-precision BESSEL function of the first kind of order 0 of X, where X is a double-precision argument.
dbesj1	REAL*8	Computes the double-precision BESSEL function of the first kind of order 1 of X, where X is a double-precision argument.
dbesjn	REAL*8	Computes the double-precision BESSEL function of the first kind of order N of X, where N is an integer and X is a double-precision argument.
dbesy0	REAL*8	Computes the double-precision BESSEL function of the second kind of order 0 of X, where X, where X is a double-precision argument.
dbesy1	REAL*8	Computes the double-precision BESSEL function of the second kind of order 1 of X, where X, where X is a double-precision argument.
dbesyn	REAL*8	Computes the double-precision BESSEL function of the second kind of order N of X, where N is an integer and X, where X is a double-precision argument.
derf	REAL*8	Computes the double-precision error function of X, where X is a double-precision argument.
derfc	REAL*8	Computes the complementary double-precision error function of X, where X is a double-precision argument.
dffrac	REAL*8	Returns fractional accuracy of a REAL*8 floating-point value.
dflmax	REAL*8	Returns the maximum positive REAL*8 floating-point value.
dflmin	REAL*8	Returns the minimum positive REAL*8 floating-point value.
drandm	REAL*8	Generates a REAL*8 random number.
dsecnds	REAL*8	Returns the number of real time seconds since midnight minus the supplied argument value.
dtime	REAL*4	Returns the elapsed user and system time in seconds since the last call to dtime.
erf	REAL*4	Computes the error function of X, where X is Real.

Routine	Result	Description
erfc	REAL	Computes the complementary error function of X, where X is Real.
etime	REAL*4	Returns the elapsed time in seconds since the start of program execution.
exit	None	Immediately terminates the program and passes a status to the parent process.
fdate	STRING	Returns the current date and time as an ASCII string.
ffrac	REAL*4	Returns the fractional accuracy of a REAL*4 floating-point value.
fgetc	INTEGER*4	Gets a character or word from an input stream. Returns the next byte or and integer
flmax	REAL*4	Returns the maximum positive REAL*4 floating-point value.
flmin	REAL*4	Returns the minimum positive REAL*4 floating-point value.
flush	None	Writes the output to a logical unit.
fputc	INTEGER*4	Writes a character or word from an input stream to a logical unit. Returns 0 if successful or an error.
free	None	Frees memory previously allocated by MALLOC(). Intended for users compiling legacy code. Use DEALLOCATE for newer code.
fseek	INTEGER*4	Repositions the file pointer associated with the specified file. Returns 0 if successful, 1 otherwise.
fseek64	INTEGER*4	Repositions the file pointer associated with the specified stream. Returns 0 if successful, 1 otherwise.
fstat	INTEGER*4	Returns file status information about the referenced open file or shared memory object.
fstat64	INTEGER*4	Returns information in a 64-bit structure about the referenced open file or shared memory object.
ftell	INTEGER*4	Returns the current value of the file pointer associated with the specified stream.
ftell64	INTEGER*8	Returns the current value of the file pointer associated with the specified stream.
gerror	STRING	Writes system error messages.
getarg	STRING	Returns the list of parameters that were passed to the current process when it was started.
getc	INTEGER*4	Retrieves the character at the front of the specified character list, or -1 if empty
getcwd	INTEGER*4	Retrieves the pathname of the current working directory or null if fails.

Routine	Result	Description
getenv		Returns the value of the specified environment variable(s).
getfd	INTEGER*4	Returns the file descriptor associated with a Fortran logical unit.
getgid	INTEGER*4	Returns the numerical group ID of the current process.
getlog	STRING	Stores the user's login name in NAME. If the login name is not found, then NAME is filled with blanks.
getpid	INTEGER*4	Returns the process numerical identifier of the current process.
getuid	INTEGER*4	Returns the numerical user ID of the current process.
gmtime	INTEGER*4	Converts and returns the date and time formats to GM (Greenwich) time as month, day, and so on.
hostnm	INTEGER*4	Sets or Gets the name of the current host. If setting the hostname, returns 0 if successful, errno if not.
iargc	INTEGER*4	Returns an integer representing the number of arguments for the last program entered on the command line.
idate	INTEGER*4	Returns the date in numerical form, day, month, year.
ierrno	INTEGER*4	Returns the system error number for the last error.
inmax	INTEGER*4	Returns the maximum positive integer value.
ioinit	None	Establishes the properties of file I/O for files opened after the call to ioinit, such as whether to recognize carriage control, how to treat blanks and zeros, and whether to open files at the beginning or end of the file.
irand1	INTEGER*4	Generates pseudo-random integer in the range of 0 through (2**31)-1, or (2**15)-1 if called with no argument.
irand2	INTEGER*4	Generates pseudo-random integer in the range of 0 through (2**31)-1, or (2**15)-1 if called with no argument.
irandm	INTEGER*4	Generates pseudo-random integer in the range of 0 through (2**31)-1, or (2**15)-1 if called with no argument.
isatty	LOGICAL	Finds the name of a terminal port. Returns TRUE if the specified unit is a terminal.
itime	numerical form of time	Fills and returns TARRAY with numerical values at the current local time, with elements 1,2,and 3 of TARRY being the hour (1-24), minute (1-60) and seconds (1-60).
kill	INTEGER*4	Sends the specified signal to the specified process or group of processes. Returns 0 if successful, -1 otherwise
link	INTEGER*4	Creates an additional directory entry for the specified existing file.

Routine	Result	Description
lnblnk	INTEGER*4	Returns the position of the last non-blank string character in the specified string.
loc	INTEGER*4	Returns the address of an object.
long	INTEGER*4	Converts INTEGER*2 to INTEGER*4
lstat	INTEGER*4	Obtains information about the referenced open file or shared memory object in a large-file enables programming environment.
lstat64	INTEGER*4	Obtains information in a 64-bit structure about the referenced open file or shared memory object in a large-file enables programming environment.
ltime	Array of INTEGER*4	Converts the system time from seconds into TARRAY, which contains GMT for the current local time zone.
malloc	INTEGER*8	Allocates SIZE byes of dynamic memory, returning the address of the allocated memory. Intended for users compiling legacy code. Use ALLOCATE for newer code.
mclock	INTEGER*4	Returns time accounting information about the current process and its child processes in 1/100 or second units of measure. The returned value is the sum of the current process's user time and system time of all child processes.
outstr	INTEGER*4	Outputs the value of the specified character to the standard output file.
perror	None	Writes a message to standard error output the describes the last error encountered by a system call or library subroutine.
putc	INTEGER*4	Puts the specified character at the end of the character list.
putenv	INTEGER*4	Sets the value of the specified environment variable or creates a new environment variable.
qsort	INTEGER*4	Uses quick-sort algorithm to sort a table of data.
rand1	REAL*4	Provides a method for generating a random number that can be used as the starting point for the rand procedure.
rand2	REAL*4	Provides a random value between 0 and 1, which is generated using the specified seed value, and computed for each returned row when used in the select list.
random	REAL*4	Uses a non-linear additive feedback random-number generator to return pseudo-random numbers in the range of 0 to $(2^{31}-1)$
rename	INTEGER*4	Renames the specified directory or file

Routine	Result	Description
rindex	INTEGER*4	Returns the index of the last occurrence of a specific string of characters in a specified string.
rtc	REAL*8	Returns the real-time clock value expressed as a number of clock ticks.
secnds	REAL*4	Gets the time in seconds from the real-time system clock. If the value is zero, the time in seconds from midnight is used.
short	INTEGER*2	Converts INTEGER*4 to INTEGER*2.
signal	INTEGER*4	Specifies the action to take upon delivery of a signal.
sleep	None	Puts the calling kernel thread to sleep, requiring it to wait for a wakeup to be issued to continue to run. Provided for compatibility with older code and should not be used with new code.
srand1	None	Sets the seed for the pseudo-random number generation that rand1 provides.
srand2	None	Sets the seed for the pseudo-random number generation that rand2 provides.
stat	INTEGER*4	Obtains information about the specified file.
stat64	INTEGER*4	Obtains information in a 64-bit structure about the specified file.
stime	INTEGER*4	Sets the current value of the specified parameter for the system-wide timer.
symlnk	INTEGER*4	Creates a symbolic link with the specified name to the specified file.
system	INTEGER*4	Runs a shell command.
time	INTEGER*4	Returns the time in seconds since January 1, 1970.
timef	REAL*8	Returns the elapsed time in milliseconds since the first call to timef.
times	INTEGER*4	Fills the specified structure with time-accounting information.
ttynam	STRING(100)	Either gets the path name of the terminal or determines if the device is a terminal.
unlink	INTEGER*4	Removes the specified directory entry, and decreases the link count of the file referenced by the link.

Routine	Result	Description
wait	INTEGER*4	Suspends the calling thread until the process receives a signal that is not blocked or ignored, or until the calling process' child processes stop or terminate.

Using the DFWIN module

The DFWIN module includes all the modules needed to access the Win32 API. You can use modules supporting specific portions of the Win32 API separately. DFWIN is the only module you need to access the Fortran interfaces to the Win32 API. To use this module, add the following line to your Fortran code.

```
use dfwin
```

To utilize any of the Win32 API interfaces, you can add a Fortran use statement for the specific library or module that includes it. For example, to use user32.lib, add the following Fortran use statement:

```
use user32
```

For information on the arguments and functionality of a given routine, refer to The Microsoft Windows API documentation. The function calls made through the module interfaces ultimately resolve to C Language interfaces, so some accommodation for inter-language calling conventions must be made in the Fortran application. These accommodations include:

- On x64 platforms, pointers and pointer types such as HANDLE, HINSTANCE, WPARAM, and HWND must be treated as 8-byte quantities (INTEGER(8)). On x86 (32-bit) platforms, these are 4-byte quantities (INTEGER(4)).
- In general, C makes calls by value while Fortran makes calls by reference.
- When doing Windows development one must sometimes provide callback functions for message processing, dialog processing, etc. These routines are called by the Windows system when events are processed. To provide the expected function signature for a callback function, the user may need to use the STDCALL attribute directive (!DEC$ ATTRIBUTE::STDCALL) in the declaration.

Supported Libraries and Modules

The following tables provide lists of the functions in each library or module that PGI supports in DFWIN.

Note

For information on the interfaces associated with these functions, refer to the files located here:

```
C:\Program Files\PGI\win32\9.0-0\src
```

or

```
C:\Program Files\PGI\win64\9.0-0\src
```

advapi32

The following table lists the functions that advapi32 includes:

Table 23.4. DFWIN advapi32 Functions

AccessCheckAndAuditAlarm	AccessCheckByType
AccessCheckByTypeAndAuditAlarm	AccessCheckByTypeResultList
AccessCheckByTypeResultListAndAuditAlarm	AccessCheckByTypeResultListAndAuditAlarmByHandle
AddAccessAllowedAce	AddAccessAllowedAceEx
AddAccessAllowedObjectAce	AddAccessDeniedAce
AddAccessDeniedAceEx	AddAccessDeniedObjectAce
AddAce	AddAuditAccessAce
AddAuditAccessAceEx	AddAuditAccessObjectAce
AdjustTokenGroups	AdjustTokenPrivileges
AllocateAndInitializeSid	AllocateLocallyUniqueId
AreAllAccessesGranted	AreAnyAccessesGranted
BackupEventLog	CheckTokenMembership
ClearEventLog	CloseEncryptedFileRaw
CloseEventLog	ConvertToAutoInheritPrivateObjectSecurity
CopySid	CreatePrivateObjectSecurity
CreatePrivateObjectSecurityEx	CreatePrivateObjectSecurityWithMultipleInheritance
CreateProcessAsUser	CreateProcessWithLogonW
CreateProcessWithTokenW	CreateRestrictedToken
CreateWellKnownSid	DecryptFile
DeleteAce	DeregisterEventSource
DestroyPrivateObjectSecurity	DuplicateToken
DuplicateTokenEx	EncryptFile
EqualDomainSid	EqualPrefixSid
EqualSid	FileEncryptionStatus
FindFirstFreeAce	FreeSid
GetAce	GetAclInformation
GetCurrentHwProfile	GetEventLogInformation
GetFileSecurity	GetKernelObjectSecurity
GetLengthSid	GetNumberOfEventLogRecords
GetOldestEventLogRecord	GetPrivateObjectSecurity
GetSecurityDescriptorControl	GetSecurityDescriptorDacl
GetSecurityDescriptorGroup	GetSecurityDescriptorLength
GetSecurityDescriptorOwner	GetSecurityDescriptorRMControl
GetSecurityDescriptorSacl	GetSidIdentifierAuthority

GetSidLengthRequired
GetSidSubAuthority
GetSidSubAuthorityCount
GetTokenInformation
GetUserName
GetWindowsAccountDomainSid
ImpersonateAnonymousToken
ImpersonateLoggedOnUser
ImpersonateNamedPipeClient
ImpersonateSelf
InitializeAcl
InitializeSecurityDescriptor
InitializeSid
IsTextUnicode
IsTokenRestricted
IsTokenUntrusted
IsValidAcl
IsValidSecurityDescriptor
IsValidSid
IsWellKnownSid
LogonUser
LogonUserEx
LookupAccountName
LookupAccountSid
LookupPrivilegeDisplayName
LookupPrivilegeName
LookupPrivilegeValue
MakeAbsoluteSD
MakeAbsoluteSD2
MakeSelfRelativeSD
MapGenericMask
NotifyChangeEventLog
ObjectCloseAuditAlarm
ObjectDeleteAuditAlarm
ObjectOpenAuditAlarm
ObjectPrivilegeAuditAlarm
OpenBackupEventLog
OpenEncryptedFileRaw
OpenEventLog
OpenProcessToken
OpenThreadToken
PrivilegeCheck
PrivilegedServiceAuditAlarm
ReadEncryptedFileRaw
ReadEventLog
RegisterEventSource
ReportEvent
RevertToSelf
SetAclInformation
SetFileSecurity
SetKernelObjectSecurity
SetPrivateObjectSecurity
SetPrivateObjectSecurityEx
SetSecurityDescriptorControl
SetSecurityDescriptorDacl
SetSecurityDescriptorGroup
SetSecurityDescriptorOwner
SetSecurityDescriptorRMControl
SetSecurityDescriptorSacl
SetThreadToken
SetTokenInformation
WriteEncryptedFileRaw

comdlg32

The following table lists the functions that `comdlg32` includes:

AfxReplaceText
ChooseColor
ChooseFont
CommDlgExtendedError
FindText
GetFileTitle

GetOpenFileName	GetSaveFileName	PageSetupDlg
PrintDlg	PrintDlgEx	ReplaceText

dfwbase

These are the functions that `dfwbase` includes:

chartoint	LoByte	MakeWord
chartoreal	LoWord	MakeWparam
CopyMemory	LoWord64	PaletteIndex
GetBlueValue	MakeIntAtom	PaletteRGB
GetGreenValue	MakeIntResource	PrimaryLangID
GetRedValue	MakeLangID	RGB
HiByte	MakeLCID	RtlCopyMemory
HiWord	MakeLong	SortIDFromLCID
HiWord64	MakeLParam	SubLangID
inttochar	MakeLResult	

dfwinty

These are the functions that `dfwinty` includes:

dwNumberOfFunctionKeys	rdFunction

gdi32

These are the functions that `gdi32` includes:

AbortDoc	AbortPath	AddFontMemResourceEx
AddFontResource	AddFontResourceEx	AlphaBlend
AngleArc	AnimatePalette	Arc
ArcTo	BeginPath	BitBlt
CancelDC	CheckColorsInGamut	ChoosePixelFormat
Chord	CloseEnhMetaFile	CloseFigure
CloseMetaFile	ColorCorrectPalette	ColorMatchToTarget
CombineRgn	CombineTransform	CopyEnhMetaFile
CopyMetaFile	CreateBitmap	CreateBitmapIndirect
CreateBrushIndirect	CreateColorSpace	CreateCompatibleBitmap
CreateCompatibleDC	CreateDC	CreateDIBitmap
CreateDIBPatternBrush	CreateDIBPatternBrushPt	CreateDIBSection
CreateDiscardableBitmap	CreateEllipticRgn	CreateEllipticRgnIndirect
CreateEnhMetaFile	CreateFont	CreateFontIndirect

CreateFontIndirectEx
CreateHalftonePalette
CreateHatchBrush
CreateIC
CreateMetaFile
CreatePalette
CreatePatternBrush
CreatePen
CreatePenIndirect
CreatePolygonRgn
CreatePolyPolygonRgn
CreateRectRgn
CreateRectRgnIndirect
CreateRoundRectRgn
CreateScalableFontResource
CreateSolidBrush
DeleteColorSpace
DeleteDC
DeleteEnhMetaFile
DeleteMetaFile
DeleteObject
DescribePixelFormat
DeviceCapabilities
DPtoLP
DrawEscape
Ellipse
EndDoc
EndPage
EndPath
EnumEnhMetaFile
EnumFontFamilies
EnumFontFamiliesEx
EnumFonts
EnumICMProfiles
EnumMetaFile
EnumObjects
EqualRgn
Escape
ExcludeClipRect
ExtCreatePen
ExtCreateRegion
ExtEscape
ExtFloodFill
ExtSelectClipRgn
ExtTextOut
FillPath
FillRgn
FixBrushOrgEx
FlattenPath
FloodFill
FrameRgn
GdiComment
GdiFlush
GdiGetBatchLimit
GdiSetBatchLimit
GetArcDirection
GetAspectRatioFilterEx
GetBitmapBits
GetBitmapDimensionEx
GetBkColor
GetBkMode
GetBoundsRect
GetBrushOrgEx
GetCharABCWidthsA
GetCharABCWidthsFloat
GetCharABCWidthsI
GetCharABCWidthsW
GetCharacterPlacement
GetCharWidth
GetCharWidth32
GetCharWidthFloat
GetCharWidthI
GetClipBox
GetClipRgn
GetColorAdjustment
GetColorSpace
GetCurrentObject
GetCurrentPositionEx
GetDCBrushColor
GetDCOrgEx
GetDCPenColor
GetDeviceCaps
GetDeviceGammaRamp
GetDIBColorTable
GetDIBits
GetEnhMetaFile
GetEnhMetaFileBits
GetEnhMetaFileDescriptionA
GetEnhMetaFileDescriptionW
GetEnhMetaFileHeader
GetEnhMetaFilePaletteEntries
GetEnhMetaFilePixelFormat
GetFontData
GetFontLanguageInfo
GetFontUnicodeRanges
GetGlyphIndices
GetGlyphOutline
GetGraphicsMode
GetICMProfileA
GetICMProfileW
GetKerningPairs
GetLayout
GetLogColorSpace
GetMapMode
GetMetaFile

GetMetaFileBitsEx
GetMetaRgn
GetMiterLimit
GetNearestColor
GetNearestPaletteIndex
GetObject
GetObjectType
GetOutlineTextMetrics
GetPaletteEntries
GetPath
GetPixel
GetPixelFormat
GetPolyFillMode
GetRandomRgn
GetRasterizerCaps
GetRegionData
GetRgnBox
GetROP2
GetStockObject
GetStretchBltMode
GetSystemPaletteEntries
GetSystemPaletteUse
GetTextAlign
GetTextCharacterExtra
GetTextCharset
GetTextCharsetInfo
GetTextColor
GetTextExtentExPoint
GetTextExtentExPointI
GetTextExtentPoint
GetTextExtentPoint32
GetTextExtentPointI
GetTextFace
GetTextMetrics
GetViewportExtEx
GetViewportOrgEx
GetWindowExtEx
GetWindowOrgEx
GetWinMetaFileBits
GetWorldTransform
GradientFill
IntersectClipRect
InvertRgn
LineDD
LineTo
LPtoDP
MaskBlt
ModifyWorldTransform
MoveToEx
OffsetClipRgn
OffsetRgn
OffsetViewportOrgEx
OffsetWindowOrgEx
PaintRgn
PatBlt
PathToRegion
Pie
PlayEnhMetaFile
PlayEnhMetaFileRecord
PlayMetaFile
PlayMetaFileRecord
PlgBlt
PolyBezier
PolyBezierTo
PolyDraw
Polygon
Polyline
PolylineTo
PolyPolygon
PolyPolyline
PolyTextOut
PtInRegion
PtVisible
RealizePalette
Rectangle
RectInRegion
RectVisible
RemoveFontMemResourceEx
RemoveFontResource
RemoveFontResourceEx
ResetDC
ResizePalette
RestoreDC
RoundRect
SaveDC
ScaleViewportExtEx
ScaleWindowExtEx
SelectClipPath
SelectClipRgn
SelectObject
SelectPalette
SetAbortProc
SetArcDirection
SetBitmapBits
SetBitmapDimensionEx
SetBkColor
SetBkMode
SetBoundsRect
SetBrushOrgEx
SetColorAdjustment
SetColorSpace
SetDCBrushColor
SetDCPenColor
SetDeviceGammaRamp
SetDIBColorTable

SetDIBits
SetDIBitsToDevice
SetEnhMetaFileBits
SetGraphicsMode
SetICMMode
SetICMProfile
SetLayout
SetMapMode
SetMapperFlags
SetMetaFileBitsEx
SetMetaRgn
SetMiterLimit
SetPaletteEntries
SetPixel
SetPixelFormat
SetPixelV
SetPolyFillMode
SetRectRgn
SetROP2
SetStretchBltMode
SetSystemPaletteUse
SetTextAlign
SetTextCharacterExtra
SetTextColor
SetTextJustification
SetViewportExtEx
SetViewportOrgEx
SetWindowExtEx
SetWindowOrgEx
SetWinMetaFileBits
SetWorldTransform
StartDoc
StartPage
StretchBlt
StretchDIBits
StrokeAndFillPath
StrokePath
SwapBuffers
TextOut
TranslateCharsetInfo
TransparentBlt
UnrealizeObject
UpdateColors
UpdateICMRegKey
wglCopyContext
wglCreateContext
wglCreateLayerContext
wglDeleteContext
wglDescribeLayerPlane
wglGetCurrentContext
wglGetCurrentDC
wglGetLayerPaletteEntries
wglGetProcAddress
wglMakeCurrent
wglRealizeLayerPalette
wglSetLayerPaletteEntries
wglShareLists
wglSwapLayerBuffers
wglSwapMultipleBuffers
wglUseFontBitmaps
wglUseFontOutlines
WidenPath

kernel32

These are the functions that `kernel32` includes:

ActivateActCtx
AddAtom
AddConsoleAlias
AddRefActCtx
AddVectoredContinueHandler
AddVectoredExceptionHandler
AllocateUserPhysicalPages
AllocConsole
AreFileApisANSI
AssignProcessToJobObject
AttachConsole
BackupRead
BackupSeek
BackupWrite
Beep
BeginUpdateResource
BindIoCompletionCallback
BuildCommDCB
BuildCommDCBAndTimeouts
CallNamedPipe
CancelDeviceWakeupRequest
CancelIo
CancelTimerQueueTimer
CancelWaitableTimer

CheckNameLegalDOS8Dot3
CheckRemoteDebuggerPresent
ClearCommBreak
ClearCommError
CloseHandle
CommConfigDialog
CompareFileTime
ConnectNamedPipe
ContinueDebugEvent
ConvertFiberToThread
ConvertThreadToFiber
ConvertThreadToFiberEx
CopyFile
CopyFileEx
CreateActCtx
CreateConsoleScreenBuffer
CreateDirectory
CreateDirectoryEx
CreateEvent
CreateFiber
CreateFiberEx
CreateFile
CreateFileMapping
CreateHardLink
CreateIoCompletionPort
CreateJobObject
CreateJobSet
CreateMailslot
CreateMemoryResourceNotification
CreateMutex
CreateNamedPipe
CreatePipe
CreateProcess
CreateRemoteThread
CreateSemaphore
CreateTapePartition
CreateThread
CreateTimerQueue
CreateTimerQueueTimer
CreateWaitableTimer
DeactivateActCtx
DebugActiveProcess
DebugActiveProcessStop
DebugBreak
DebugBreakProcess
DebugSetProcessKillOnExit
DecodePointer
DecodeSystemPointer
DefineDosDevice
DeleteAtom
DeleteCriticalSection
DeleteFiber
DeleteFile
DeleteTimerQueue
DeleteTimerQueueEx
DeleteTimerQueueTimer
DeleteVolumeMountPoint
DeviceIoControl
DisableThreadLibraryCalls
DisconnectNamedPipe
DnsHostnameToComputerName
DosDateTimeToFileTime
DuplicateHandle
EncodePointer
EncodeSystemPointer
EndUpdateResource
EnterCriticalSection
EnumResourceLanguages
EnumResourceNames
EnumResourceTypes

EnumSystemFirmwareTables
EscapeCommFunction
ExitThread
FatalAppExit
FileTimeToDosDateTime
FileTimeToSystemTime
FillConsoleOutputCharacter
FindActCtxSectionString
FindClose
FindFirstChangeNotification
FindFirstFileEx
FindFirstVolumeMountPoint
FindNextFile
FindNextVolumeMountPoint
FindResourceEx
FindVolumeMountPointClose
FlsFree
FlsSetValue
FlushFileBuffers
FlushViewOfFile
FreeConsole
FreeLibrary
FreeResource
GenerateConsoleCtrlEvent
GetBinaryType
GetCommConfig
GetCommModemStatus
GetCommState
GetCompressedFileSize
GetConsoleAlias
GetConsoleAliasesLength
GetConsoleAliasExesLength
GetConsoleCursorInfo
GetConsoleFontSize
GetConsoleOutputCP
EraseTape
ExitProcess
ExpandEnvironmentStrings
FatalExit
FileTimeToLocalFileTime
FillConsoleOutputAttribute
FindActCtxSectionGuid
FindAtom
FindCloseChangeNotification
FindFirstFile
FindFirstVolume
FindNextChangeNotification
FindNextVolume
FindResource
FindVolumeClose
FlsAlloc
FlsGetValue
FlushConsoleInputBuffer
FlushInstructionCache
FormatMessage
FreeEnvironmentStrings
FreeLibraryAndExitThread
FreeUserPhysicalPages
GetAtomName
GetCommandLine
GetCommMask
GetCommProperties
GetCommTimeouts
GetComputerName
GetConsoleAliases
GetConsoleAliasExes
GetConsoleCP
GetConsoleDisplayMode
GetConsoleMode
GetConsoleProcessList

GetConsoleScreenBufferInfo
GetConsoleSelectionInfo
GetConsoleTitle
GetConsoleWindow
GetCurrentActCtx
GetCurrentConsoleFont
GetCurrentDirectory
GetCurrentProcess
GetCurrentProcessId
GetCurrentProcessorNumber
GetCurrentThread
GetCurrentThreadId
GetDefaultCommConfig
GetDevicePowerState
GetDiskFreeSpace
GetDiskFreeSpaceEx
GetDllDirectory
GetDriveType
GetEnvironmentStrings
GetEnvironmentVariable
GetExitCodeProcess
GetExitCodeThread
GetFileAttributes
GetFileAttributesEx
GetFileInformationByHandle
GetFileSize
GetFileSizeEx
GetFileTime
GetFileType
GetFirmwareEnvironmentVariable
GetFullPathName
GetHandleInformation
GetLargePageMinimum
GetLargestConsoleWindowSize
GetLastError
GetLocalTime
GetLogicalDrives
GetLogicalDriveStrings
GetLogicalProcessorInformation
GetLongPathName
GetMailslotInfo
GetModuleFileName
GetModuleHandle
GetModuleHandleEx
GetNamedPipeHandleState
GetNamedPipeInfo
GetNativeSystemInfo
GetNumaAvailableMemoryNode
GetNumaHighestNodeNumber
GetNumaNodeProcessorMask
GetNumaProcessorNode
GetNumberOfConsoleInputEvents
GetNumberOfConsoleMouseButtons
GetOverlappedResult
GetPriorityClass
GetPrivateProfileInt
GetPrivateProfileSection
GetPrivateProfileSectionNames
GetPrivateProfileString
GetPrivateProfileStruct
GetProcAddress
GetProcessAffinityMask
GetProcessHandleCount
GetProcessHeap
GetProcessHeaps
GetProcessId
GetProcessIdOfThread
GetProcessIoCounters
GetProcessPriorityBoost
GetProcessShutdownParameters

GetProcessTimes
GetProcessVersion
GetProcessWorkingSetSize
GetProcessWorkingSetSizeEx
GetProfileInt
GetProfileSection
GetProfileString
GetQueuedCompletionStatus
GetShortPathName
GetStartupInfo
GetStdHandle
GetSystemDirectory
GetSystemFirmwareTable
GetSystemInfo
GetSystemRegistryQuota
GetSystemTime
GetSystemTimeAdjustment
GetSystemTimeAsFileTime
GetSystemWindowsDirectory
GetSystemWow64Directory
GetTapeParameters
GetTapePosition
GetTapeStatus
GetTempFileName
GetTempPath
GetThreadContext
GetThreadId
GetThreadIOPendingFlag
GetThreadPriority
GetThreadPriorityBoost
GetThreadSelectorEntry
GetThreadTimes
GetTickCount
GetTimeZoneInformation
GetVersion
GetVersionEx
GetVolumeInformation
GetVolumeNameForVolumeMountPoint
GetVolumePathName
GetVolumePathNamesForVolumeName
GetWindowsDirectory
GetWriteWatch
GlobalAddAtom
GlobalAlloc
GlobalCompact
GlobalDeleteAtom
GlobalFindAtom
GlobalFix
GlobalFlags
GlobalFree
GlobalGetAtomName
GlobalHandle
GlobalLock
GlobalMemoryStatus
GlobalMemoryStatusEx
GlobalReAlloc
GlobalSize
GlobalUnfix
GlobalUnlock
GlobalUnWire
GlobalWire
HeapAlloc
HeapCompact
HeapCreate
HeapDestroy
HeapFree
HeapLock
HeapQueryInformation
HeapReAlloc
HeapSetInformation

HeapSize
HeapUnlock
HeapValidate
HeapWalk
InitAtomTable
InitializeCriticalSection
InitializeCriticalSectionAndSpinCount
InitializeSListHead
InterlockedCompareExchange
InterlockedCompareExchange64
InterlockedDecrement
InterlockedExchange
InterlockedExchangeAdd
InterlockedFlushSList
InterlockedIncrement
InterlockedPopEntrySList
InterlockedPushEntrySList
IsBadCodePtr
IsBadHugeReadPtr
IsBadHugeWritePtr
IsBadReadPtr
IsBadStringPtr
IsBadWritePtr
IsDebuggerPresent
IsProcessInJob
IsProcessorFeaturePresent
IsSystemResumeAutomatic
LeaveCriticalSection
LoadLibrary
LoadLibraryEx
LoadModule
LoadResource
LocalAlloc
LocalCompact
LocalFileTimeToFileTime
LocalFlags
LocalFree
LocalHandle
LocalLock
LocalReAlloc
LocalShrink
LocalSize
LocalUnlock
LockFile
LockFileEx
LockResource
lstrcat
lstrcmp
lstrcmpi
lstrcpy
lstrcpyn
lstrlen
MapUserPhysicalPages
MapUserPhysicalPagesScatter
MapViewOfFile
MapViewOfFileEx
MoveFile
MoveFileEx
MoveFileWithProgress
MulDiv
NeedCurrentDirectoryForExePath
OpenEvent
OpenFile
OpenFileMapping
OpenJobObject
OpenMutex
OpenProcess
OpenSemaphore
OpenThread
OpenWaitableTimer

OutputDebugString
PeekNamedPipe
PrepareTape
PulseEvent
QueryActCtxW
QueryDosDevice
QueryMemoryResourceNotification
QueryPerformanceFrequency
QueueUserWorkItem
ReadConsole
ReadConsoleOutput
ReadConsoleOutputCharacter
ReadFile
ReadFileScatter
RegisterWaitForSingleObject
ReleaseActCtx
ReleaseSemaphore
RemoveVectoredContinueHandler
ReOpenFile
RequestDeviceWakeup
ResetEvent
RestoreLastError
ScrollConsoleScreenBuffer
SetCommBreak
SetCommMask
SetCommTimeouts
SetComputerNameEx
SetConsoleCP
SetConsoleCursorInfo
SetConsoleMode
SetConsoleScreenBufferSize
SetConsoleTitle
SetCriticalSectionSpinCount
SetDefaultCommConfig
SetEndOfFile
PeekConsoleInput
PostQueuedCompletionStatus
ProcessIdToSessionId
PurgeComm
QueryDepthSList
QueryInformationJobObject
QueryPerformanceCounter
QueueUserAPC
RaiseException
ReadConsoleInput
ReadConsoleOutputAttribute
ReadDirectoryChangesW
ReadFileEx
ReadProcessMemory
RegisterWaitForSingleObjectEx
ReleaseMutex
RemoveDirectory
RemoveVectoredExceptionHandler
ReplaceFile
RequestWakeupLatency
ResetWriteWatch
ResumeThread
SearchPath
SetCommConfig
SetCommState
SetComputerName
SetConsoleActiveScreenBuffer
SetConsoleCtrlHandler
SetConsoleCursorPosition
SetConsoleOutputCP
SetConsoleTextAttribute
SetConsoleWindowInfo
SetCurrentDirectory
SetDllDirectory
SetEnvironmentStrings

SetEnvironmentVariable
SetEvent
SetFileApisToOEM
SetFilePointer
SetFileShortName
SetFileValidData
SetHandleCount
SetInformationJobObject
SetLocalTime
SetMessageWaitingIndicator
SetPriorityClass
SetProcessPriorityBoost
SetProcessWorkingSetSize
SetStdHandle
SetSystemTimeAdjustment
SetTapePosition
SetThreadContext
SetThreadIdealProcessor
SetThreadPriorityBoost
SetTimerQueueTimer
SetUnhandledExceptionFilter
SetVolumeLabel
SetWaitableTimer
SizeofResource
SleepEx
SwitchToFiber
SystemTimeToFileTime
TerminateJobObject
TerminateThread
TlsFree
TlsSetValue
TransmitCommChar
TzSpecificLocalTimeToSystemTime
UnlockFile
UnmapViewOfFile
SetErrorMode
SetFileApisToANSI
SetFileAttributes
SetFilePointerEx
SetFileTime
SetFirmwareEnvironmentVariable
SetHandleInformation
SetLastError
SetMailslotInfo
SetNamedPipeHandleState
SetProcessAffinityMask
SetProcessShutdownParameters
SetProcessWorkingSetSizeEx
SetSystemTime
SetTapeParameters
SetThreadAffinityMask
SetThreadExecutionState
SetThreadPriority
SetThreadStackGuarantee
SetTimeZoneInformation
SetupComm
SetVolumeMountPoint
SignalObjectAndWait
Sleep
SuspendThread
SwitchToThread
SystemTimeToTzSpecificLocalTime
TerminateProcess
TlsAlloc
TlsGetValue
TransactNamedPipe
TryEnterCriticalSection
UnhandledExceptionFilter
UnlockFileEx
UnregisterWait

UnregisterWaitEx
UpdateResource
VerifyVersionInfo
VirtualAlloc
VirtualAllocEx
VirtualFree
VirtualFreeEx
VirtualLock
VirtualProtect
VirtualProtectEx
VirtualQuery
VirtualQueryEx
VirtualUnlock
WaitCommEvent
WaitForDebugEvent
WaitForMultipleObjects
WaitForMultipleObjectsEx
WaitForSingleObject
WaitForSingleObjectEx
WaitNamedPipe
WinExec
Wow64DisableWow64FsRedirection
Wow64EnableWow64FsRedirection
Wow64RevertWow64FsRedirection
WriteConsole
WriteConsoleInput
WriteConsoleOutput
WriteConsoleOutputAttribute
WriteConsoleOutputCharacter
WriteFile
WriteFileEx
WriteFileGather
WritePrivateProfileSection
WritePrivateProfileString
WritePrivateProfileStruct
WriteProcessMemory
WriteProfileSection
WriteProfileString
WriteTapemark
WTSGetActiveConsoleSessionId
ZombifyActCtx
_hread
_hwrite
_lclose
_lcreat
_llseek
_lopen
_lread
_lwrite

shell32

These are the functions that `shell32` includes:

DoEnvironmentSubst
ShellExecuteEx
DragAcceptFiles
Shell_NotifyIcon
DragFinish
SHEmptyRecycleBin
DragQueryFile
SHFileOperation
DragQueryPoint
SHFreeNameMappings
DuplicateIcon
SHGetDiskFreeSpaceEx
ExtractAssociatedIcon
SHGetFileInfo
ExtractIcon
SHGetNewLinkInfo

ExtractIconEx
FindExecutable
IsLFNDrive
SHAppBarMessage
SHCreateProcessAsUserW
ShellAbout
ShellExecute
SHInvokePrinterCommand
SHIsFileAvailableOffline
SHLoadNonloadedIconOverlayIdentifiers
SHQueryRecycleBin
SHSetLocalizedName
WinExecError

user32

These are the functions that `user32` includes:

ActivateKeyboardLayout
AdjustWindowRect
AdjustWindowRectEx
AllowSetForegroundWindow
AnimateWindow
AnyPopup
AppendMenu
ArrangeIconicWindows
AttachThreadInput
BeginDeferWindowPos
BeginPaint
BringWindowToTop
BroadcastSystemMessage
BroadcastSystemMessageEx
CallMsgFilter
CallNextHookEx
CallWindowProc
CascadeWindows
ChangeClipboardChain
ChangeDisplaySettings
ChangeDisplaySettingsEx
ChangeMenu
CharLower
CharLowerBuff
CharNext
CharNextEx
CharPrev
CharPrevEx
CharToOem
CharToOemBuff
CharUpper
CharUpperBuff
CheckDlgButton
CheckMenuItem
CheckMenuRadioItem
CheckRadioButton
ChildWindowFromPoint
ChildWindowFromPointEx
ClientToScreen
ClipCursor
CloseClipboard
CloseDesktop
CloseWindow
CloseWindowStation
CopyAcceleratorTable
CopyCursor
CopyIcon
CopyImage
CopyRect
CountClipboardFormats
CreateAcceleratorTable
CreateCaret
CreateCursor
CreateDesktop
CreateDialogIndirectParam
CreateDialogParam
CreateIcon
CreateIconFromResource
CreateIconFromResourceEx
CreateIconIndirect
CreateMDIWindow
CreateMenu
CreatePopupMenu
CreateWindow
CreateWindowEx
CreateWindowStation
DeferWindowPos
DefFrameProc
DefMDIChildProc
DefRawInputProc
DefWindowProc
DeleteMenu
DeregisterShellHookWindow
DestroyAcceleratorTable
DestroyCaret
DestroyCursor
DestroyIcon
DestroyMenu

DestroyWindow
DialogBoxIndirectParam
DialogBoxParam1
DialogBoxParam2
DisableProcessWindowsGhosting
DispatchMessage
DlgDirList
DlgDirListComboBox
DlgDirSelectComboBoxEx
DlgDirSelectEx
DragDetect
DragObject
DrawAnimatedRects
DrawCaption
DrawEdge
DrawFocusRect
DrawFrameControl
DrawIcon
DrawIconIndirect
DrawMenuBar
DrawState
DrawText
DrawTextEx
EmptyClipboard
EnableMenuItem
EnableScrollBar
EnableWindow
EndDeferWindowPos
EndDialog
EndMenu
EndPaint
EndTask
EnumChildWindows
EnumClipboardFormats
EnumDesktops
EnumDesktopWindows
EnumDisplayDevices
EnumDisplayMonitors
EnumDisplaySettings
EnumDisplaySettingsEx
EnumProps
EnumPropsEx
EnumThreadWindows
EnumWindows
EnumWindowStations
EqualRect
ExcludeUpdateRgn
ExitWindowsEx
FillRect
FindWindow
FindWindowEx
FlashWindow
FlashWindowEx
FrameRect
GetActiveWindow
GetAltTabInfo
GetAncestor
GetAsyncKeyState
GetCapture
GetCaretBlinkTime
GetCaretPos
GetClassInfo
GetClassInfoEx
GetClassLong
GetClassLongPtr
GetClassName
GetClassWord
GetClientRect
GetClipboardData
GetClipboardFormatName
GetClipboardOwner
GetClipboardSequenceNumber
GetClipboardViewer
GetClipCursor
GetComboBoxInfo
GetCursor
GetCursorInfo
GetCursorPos
GetDC
GetDCEx
GetDesktopWindow
GetDialogBaseUnits
GetDlgCtrlID
GetDlgItem
GetDlgItemInt
GetDlgItemText
GetDoubleClickTime
GetFocus
GetForegroundWindow
GetGuiResources
GetGUIThreadInfo
GetIconInfo
GetInputState
GetKBCodePage
GetKeyboardLayout
GetKeyboardLayoutList
GetKeyboardLayoutName
GetKeyboardState
GetKeyboardType
GetKeyNameText
GetKeyState
GetLastActivePopup
GetLastInputInfo
GetLayeredWindowAttributes
GetListBoxInfo

GetMenu
GetMenuBarInfo
GetMenuCheckMarkDimensions
GetMenuContextHelpId
GetMenuDefaultItem
GetMenuInfo
GetMenuItemCount
GetMenuItemID
GetMenuItemInfo
GetMenuItemRect
GetMenuState
GetMenuString
GetMessage
GetMessageExtraInfo
GetMessagePos
GetMessageTime
GetMonitorInfo
GetMouseMovePointsEx
GetNextDlgGroupItem
GetNextDlgTabItem
GetOpenClipboardWindow
GetParent
GetPriorityClipboardFormat
GetProcessDefaultLayout
GetProcessWindowStation
GetProp
GetQueueStatus
GetRawInputBuffer
GetRawInputData
GetRawInputDeviceInfo
GetRawInputDeviceList
GetRegisteredRawInputDevices
GetScrollBarInfo
GetScrollInfo
GetScrollPos
GetScrollRange
GetShellWindow
GetSubMenu
GetSysColor
GetSysColorBrush
GetSystemMenu
GetSystemMetrics
GetTabbedTextExtent
GetThreadDesktop
GetTitleBarInfo
GetTopWindow
GetUpdateRect
GetUpdateRgn
GetUserObjectInformation
GetUserObjectSecurity
GetWindow
GetWindowContextHelpId
GetWindowDC
GetWindowInfo
GetWindowLong
GetWindowLongPtr
GetWindowModuleFileName
GetWindowPlacement
GetWindowRect
GetWindowRgn
GetWindowRgnBox
GetWindowText
GetWindowTextLength
GetWindowThreadProcessId
GetWindowWord
GrayString
HideCaret
HiliteMenuItem
InflateRect
InSendMessage
InSendMessageEx
InsertMenu
InsertMenuItem
InternalGetWindowText
IntersectRect
InvalidateRect
InvalidateRgn
InvertRect
IsCharAlpha
IsCharAlphaNumeric
IsCharLower
IsCharUpper
IsChild
IsClipboardFormatAvailable
IsDialogMessage
IsDlgButtonChecked
IsGUIThread
IsHungAppWindow
IsIconic
IsMenu
IsRectEmpty
IsWindow
IsWindowEnabled
IsWindowUnicode
IsWindowVisible
IsWinEventHookInstalled
IsWow64Message
IsZoomed
keybd_event
KillTimer
LoadAccelerators
LoadBitmap
LoadCursor1
LoadCursor2
LoadCursorFromFile

LoadIcon1
LoadIcon2
LoadImage
LoadKeyboardLayout
LoadMenu1
LoadMenu2
LoadMenuIndirect
LoadString
LockSetForegroundWindow
LockWindowUpdate
LockWorkStation
LookupIconIdFromDirectory
LookupIconIdFromDirectoryEx
LRESULT
MapDialogRect
MapVirtualKey
MapVirtualKeyEx
MapWindowPoints
MenuItemFromPoint
MessageBeep
MessageBox
MessageBoxEx
MessageBoxIndirect
ModifyMenu1
ModifyMenu2
MonitorFromPoint
MonitorFromRect
MonitorFromWindow
mouse_event
MoveWindow
MsgWaitForMultipleObjects
MsgWaitForMultipleObjectsEx
NotifyWinEvent
OemKeyScan
OemToChar
OemToCharBuff
OffsetRect
OpenClipboard
OpenDesktop
OpenIcon
OpenInputDesktop
OpenWindowStation
PaintDesktop
PeekMessage
PostMessage
PostQuitMessage
PostThreadMessage
PrintWindow
PrivateExtractIcons
PtInRect
RealChildWindowFromPoint
RealGetWindowClass
RedrawWindow
RegisterClass
RegisterClassEx
RegisterClipboardFormat
RegisterDeviceNotification
RegisterHotKey
RegisterRawInputDevices
RegisterShellHookWindow
RegisterWindowMessage
ReleaseCapture
ReleaseDC
RemoveMenu
RemoveProp
ReplyMessage
ScreenToClient
ScrollDC
ScrollWindow
ScrollWindowEx
SendDlgItemMessage
SendInput
SendMessage
SendMessageCallback
SendMessageTimeout
SendNotifyMessage
SetActiveWindow
SetCapture
SetCaretBlinkTime
SetCaretPos
SetClassLong
SetClassLongPtr
SetClassWord
SetClipboardData
SetClipboardViewer
SetCursor
SetCursorPos
SetDebugErrorLevel
SetDlgItemInt
SetDlgItemText
SetDoubleClickTime
SetFocus
SetForegroundWindow
SetKeyboardState
SetLastErrorEx
SetLayeredWindowAttributes
SetMenu
SetMenuContextHelpId
SetMenuDefaultItem
SetMenuInfo
SetMenuItemBitmaps
SetMenuItemInfo
SetMessageExtraInfo
SetMessageQueue
SetParent

SetProcessDefaultLayout	SetProcessWindowStation	SetProp
SetRect	SetRectEmpty	SetScrollInfo
SetScrollPos	SetScrollRange	SetSysColors
SetSystemCursor	SetThreadDesktop	SetTimer
SetUserObjectInformation	SetUserObjectSecurity	SetWindowContextHelpId
SetWindowLong	SetWindowLongPtr	SetWindowPlacement
SetWindowPos	SetWindowRgn	SetWindowsHook
SetWindowsHookEx	SetWindowText	SetWindowWord
SetWinEventHook	ShowCaret	ShowCursor
ShowOwnedPopups	ShowScrollBar	ShowWindow
ShowWindowAsync	SubtractRect	SwapMouseButton
SwitchDesktop	SwitchToThisWindow	SystemParametersInfo
TabbedTextOut	TileWindows	ToAscii
ToAsciiEx	ToUnicode	ToUnicodeEx
TrackMouseEvent	TrackPopupMenu	TrackPopupMenuEx
TranslateAccelerator	TranslateMDISysAccel	TranslateMessage
UnhookWindowsHook	UnhookWindowsHookEx	UnhookWinEvent
UnionRect	UnloadKeyboardLayout	UnregisterClass
UnregisterDeviceNotification	UnregisterHotKey	UpdateLayeredWindow
UpdateLayeredWindowIndirect	UpdateWindow	UserHandleGrantAccess
ValidateRect	ValidateRgn	VkKeyScan
VkKeyScanEx	WaitForInputIdle	WaitMessage
WindowFromDC	WindowFromPoint	WinHelp
wsprintf	wvsprintf	

winver

These are the functions that `winver` includes:

GetFileVersionInfo	VerFindFile	VerLanguageName
GetFileVersionInfoSize	VerInstallFile	VerQueryValue

wsock32

These are the functions that wsock32 includes:

accept	AcceptEx	bind
closesocket	connect	GetAcceptExSockaddrs
getpeername	gethostname	getprotobyname

getprotobynumber	getservbyname	getservbyport
getsockname	getsockopt	htonl
htons	inet_addr	inet_ntoa
ioctlsocket	listen	ntohl
ntohs	recv	select
send	sendto	setsockopt
shutdown	socket	TransmitFile
WSAAsyncGetHostByName	WSAAsyncGetProtoByName	WSAAsyncGetProtoByNumber
WSAAsyncGetServByName	WSAAsyncGetServByPort	WSAAsyncSelect
WSACancelAsyncRequest	WSACancelBlockingCall	WSACleanup
WSAGetLastError	WSAIsBlocking	WSARecvEx
WSASetBlockingHook	WSASetLastError	WSAStartup

Chapter 24. C/C++ MMX/SSE Inline Intrinsics

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

PGI provides support for MMX (and SSE/SSE2/SSE3/SSSE3/SSE4a/ABM intrinsics in C/C++ programs.

Intrinsics make the use of processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

This chapter contains these seven tables associated with inline intrinsics:

- A table of MMX inline intrinsics (mmintrin.h)
- A table of SSE inline intrinsics (xmmintrin.h)
- A table of SSE2 inline intrinsics (emmintrin.h)
- A table of SSE3 inline intrinsics (pmmintrin.h)
- A table of SSSE3 inline intrinsics (tmmintrin.h)
- A table of SSE4a inline intrinsics (ammintrin.h)
- A table of ABM inline intrinsics (intrin.h)

Using Intrinsic functions

The definitions of the intrinsics are provided in the inline library `libintrinsics.il`, which is automatically included when you compile.

Required Header File

To call these intrinsic functions from a C/C++ source, you must include the corresponding header file - one of the following:

- For MMX, use `mmintrin.h`
- For SSE, use `xmmintrin.h`
- For SSE2, use `emmintrin.h`
- For SSE3, use `pmmintrin.h`
- For SSSE3 use `tmmintrin.h`
- For SSE4a use `ammintrin.h`
- For ABM use `intrin.h`

Intrinsic Data Types

The following table describes the data types that are defined for intrinsics:

Data Types	Defined in	Description
__m64	`mmintrin.h`	For use with MMX intrinsics, this 64-bit data type stores one 64-bit or two 32-bit integer values.
__m128	`xmmintrin.h`	For use with SSE intrinsics, this 128-bit data type, aligned on 16-byte boundaries, stores four single-precision floating point values.
__m128d	`emmintrin.h`	For use with SSE2/SSE3 intrinsics, this 128-bit data type, aligned on 16-byte boundaries, stores two double-precision floating point values.
__m128i	`emmintrin.h`	For use with SSE2/SSE3 intrinsics, this 128-bit data type, aligned on 16-byte boundaries, stores two 64-bit integer values.

Intrinsic Example

The MMX/SSE intrinsics include functions for initializing variables of the types defined in the preceding table. The following sample program, `example.c`, illustrates the use of the SSE intrinsics _mm_add_ps and _mm_set_ps.

```
#include<xmmintrin.h>
 int main(){
 __m128 A, B, result;
 A = _mm_set_ps(23.3, 43.7, 234.234, 98.746); /* initialize A */
 B = _mm_set_ps(15.4, 34.3, 4.1, 8.6); /* initialize B */
 result = _mm_add_ps(A, B);
 return 0;
 }
```

To compile this program, use the following command:

```
$ pgcc example.c -o myprog
```

The inline library `libintrinsics.il` is automatically inlined.

MMX Intrinsics

PGI supports a set of MMX Intrinsics which allow the use of the MMX instructions directly from C/C++ code, without writing the assembly instructions. The following table lists the MMX intrinsics that PGI supports.

Note

Intrinsics with a * are only available on 64-bit systems.

Table 24.1. MMX Intrinsics (mmintrin.h)

_mm_empty	_m_paddd	_m_psllw	_m_pand
_m_empty	_mm_add_si64	_mm_slli_pi16	_mm_andnot_si64
_mm_cvtsi32_si64	_mm_adds_pi8	_m_psllwi	_m_pandn
_m_from_int	_m_paddsb	_mm_sll_pi32	_mm_or_si64
_mm_cvtsi64x_si64*	_mm_adds_pi16	_m_pslld	_m_por
_mm_set_pi64x*	_m_paddsw	_mm_slli_pi32	_mm_xor_si64
_mm_cvtsi64_si32	_mm_adds_pu8	_m_pslldi	_m_pxor
_m_to_int	_m_paddusb	_mm_sll_si64	_mm_cmpeq_pi8
_mm_cvtsi64_si64x*	_mm_adds_pu16	_m_psllq	_m_pcmpeqb
_mm_packs_pi16*	_m_paddusw	_mm_slli_si64	_mm_cmpgt_pi8
_m_packsswb	_mm_sub_pi8	_m_psllqi	_m_pcmpgtb
_mm_packs_pi32	_m_psubb	_mm_sra_pi16	_mm_cmpeq_pi16
_m_packssdw	_mm_sub_pi16	_m_psraw	_m_pcmpeqw
_mm_packs_pu16	_m_psubw	_mm_srai_pi16	_mm_cmpgt_pi16
_m_packuswb	_mm_sub_pi32	_m_psrawi	_m_pcmpgtw
_mm_unpackhi_pi8	_m_psubd	_mm_sra_pi32	_mm_cmpeq_pi32
_m_punpckhbw	_mm_sub_si64	_m_psrad	_m_pcmpeqd
_mm_unpackhi_pi16	_mm_subs_pi8	_mm_srai_pi32	_mm_cmpgt_pi32
_m_punpckhwd	_m_psubsb	_m_psradi	_m_pcmpgtd
_mm_unpackhi_pi32	_mm_subs_pi16	_mm_srl_pi16	_mm_setzero_si64
_m_punpckhdq	_m_psubsw	_m_psrlw	_mm_set_pi32
_mm_unpacklo_pi8	_mm_subs_pu8	_mm_srli_pi16	_mm_set_pi16
_m_punpcklbw	_m_psubusb	_m_psrlwi	_mm_set_pi8
_mm_unpacklo_pi16	_mm_subs_pu16	_mm_srl_pi32	_mm_setr_pi32
_m_punpcklwd	_m_psubusw	_m_psrld	_mm_setr_pi16
_mm_unpacklo_pi32	_mm_madd_pi16	_mm_srli_pi32	_mm_setr_pi8
_m_punpckldq	_m_pmaddwd	_m_psrldi	_mm_set1_pi32
_mm_add_pi8	_mm_mulhi_pi16	_mm_srl_si64	_mm_set1_pi16
_m_paddb	_m_pmulhw	_m_psrlq	_mm_set1_pi8
_mm_add_pi16	_mm_mullo_pi16	_mm_srli_si64	
_m_paddw	_m_pmullw	_m_psrlqi	
_mm_add_pi32	_mm_sll_pi16	_mm_and_si64	

SSE Intrinsics

PGI supports a set of SSE Intrinsics which allow the use of the SSE instructions directly from C/C++ code, without writing the assembly instructions. The following tables list the SSE intrinsics that PGI supports.

Note

Intrinsics with a * are only available on 64-bit systems.

Table 24.2. SSE Intrinsics (xmmintrin.h)

_mm_add_ss	_mm_comige_ss	_mm_load_ss
_mm_sub_ss	_mm_comineq_ss	_mm_load1_ps
_mm_mul_ss	_mm_ucomieq_ss	_mm_load_ps1
_mm_div_ss	_mm_ucomilt_ss	_mm_load_ps
_mm_sqrt_ss	_mm_ucomile_ss	_mm_loadu_ps
_mm_rcp_ss	_mm_ucomigt_ss	_mm_loadr_ps
_mm_rsqrt_ss	_mm_ucomige_ss	_mm_set_ss
_mm_min_ss	_mm_ucomineq_ss	_mm_set1_ps
_mm_max_ss	_mm_cvtss_si32	_mm_set_ps1
_mm_add_ps	_mm_cvt_ss2si	_mm_set_ps
_mm_sub_ps	_mm_cvtss_si64x*	_mm_setr_ps
_mm_mul_ps	_mm_cvtps_pi32	_mm_store_ss
_mm_div_ps	_mm_cvt_ps2pi	_mm_store_ps
_mm_sqrt_ps	_mm_cvttss_si32	_mm_store1_ps
_mm_rcp_ps	_mm_cvtt_ss2si	_mm_store_ps1
_mm_rsqrt_ps	_mm_cvttss_si64x*	_mm_storeu_ps
_mm_min_ps	_mm_cvttps_pi32	_mm_storer_ps
_mm_max_ps	_mm_cvtt_ps2pi	_mm_move_ss
_mm_and_ps	_mm_cvtsi32_ss	_mm_extract_pi16
_mm_andnot_ps	_mm_cvt_si2ss	_m_pextrw
_mm_or_ps	_mm_cvtsi64x_ss*	_mm_insert_pi16
_mm_xor_ps	_mm_cvtpi32_ps	_m_pinsrw
_mm_cmpeq_ss	_mm_cvt_pi2ps	_mm_max_pi16
_mm_cmplt_ss	_mm_movelh_ps	_m_pmaxsw
_mm_cmple_ss	_mm_setzero_ps	_mm_max_pu8
_mm_cmpgt_ss	_mm_cvtpi16_ps	_m_pmaxub
_mm_cmpge_ss	_mm_cvtpu16_ps	_mm_min_pi16
_mm_cmpneq_ss	_mm_cvtpi8_ps	_m_pminsw

_mm_cmpnlt_ss	_mm_cvtpu8_ps	_mm_min_pu8
_mm_cmpnle_ss	_mm_cvtpi32x2_ps	_m_pminub
_mm_cmpngt_ss	_mm_movehl_ps	_mm_movemask_pi8
_mm_cmpnge_ss	_mm_cvtps_pi16	_m_pmovmskb
_mm_cmpord_ss	_mm_cvtps_pi8	_mm_mulhi_pu16
_mm_cmpunord_ss	_mm_shuffle_ps	_m_pmulhuw
_mm_cmpeq_ps	_mm_unpackhi_ps	_mm_shuffle_pi16
_mm_cmplt_ps	_mm_unpacklo_ps	_m_pshufw
_mm_cmple_ps	_mm_loadh_pi	_mm_maskmove_si64
_mm_cmpgt_ps	_mm_storeh_pi	_m_maskmovq
_mm_cmpge_ps	_mm_loadl_pi	_mm_avg_pu8
_mm_cmpneq_ps	_mm_storel_pi	_m_pavgb
_mm_cmpnlt_ps	_mm_movemask_ps	_mm_avg_pu16
_mm_cmpnle_ps	_mm_getcsr	_m_pavgw
_mm_cmpngt_ps	_MM_GET_EXCEPTION_STATE	_mm_sad_pu8
_mm_cmpnge_ps	_MM_GET_EXCEPTION_MASK	_m_psadbw
_mm_cmpord_ps	_MM_GET_ROUNDING_MODE	_mm_prefetch
_mm_cmpunord_ps	_MM_GET_FLUSH_ZERO_MODE	_mm_stream_pi
_mm_comieq_ss	_mm_setcsr	_mm_stream_ps
_mm_comilt_ss	_MM_SET_EXCEPTION_STATE	_mm_sfence
_mm_comile_ss	_MM_SET_EXCEPTION_MASK	_mm_pause
_mm_comigt_ss	_MM_SET_ROUNDING_MODE	_MM_TRANSPOSE4_PS
	_MM_SET_FLUSH_ZERO_MODE	

Table 24.3 lists the SSE2 intrinsics that PGI supports and that are available in emmintrin.h.

Table 24.3. SSE2 Intrinsics (emmintrin.h)

_mm_load_sd	_mm_cmpge_sd	_mm_cvtps_pd	_mm_srl_epi32
_mm_load1_pd	_mm_cmpneq_sd	_mm_cvtsd_si32	_mm_srl_epi64
_mm_load_pd1	_mm_cmpnlt_sd	_mm_cvtsd_si64x*	_mm_slli_epi16
_mm_load_pd	_mm_cmpnle_sd	_mm_cvttsd_si32	_mm_slli_epi32
_mm_loadu_pd	_mm_cmpngt_sd	_mm_cvttsd_si64x*	_mm_slli_epi64
_mm_loadr_pd	_mm_cmpnge_sd	_mm_cvtsd_ss	_mm_srai_epi16
_mm_set_sd	_mm_cmpord_sd	_mm_cvtsi32_sd	_mm_srai_epi32
_mm_set1_pd	_mm_cmpunord_sd	_mm_cvtsi64x_sd*	_mm_srli_epi16
_mm_set_pd1	_mm_comieq_sd	_mm_cvtss_sd	_mm_srli_epi32
_mm_set_pd	_mm_comilt_sd	_mm_unpackhi_pd	_mm_srli_epi64

_mm_setr_pd
_mm_setzero_pd
_mm_store_sd
_mm_store_pd
_mm_store1_pd
_mm_store_pd1
_mm_storeu_pd
_mm_storer_pd
_mm_move_sd
_mm_add_pd
_mm_add_sd
_mm_sub_pd
_mm_sub_sd
_mm_mul_pd
_mm_mul_sd
_mm_div_pd
_mm_div_sd
_mm_sqrt_pd
_mm_sqrt_sd
_mm_min_pd
_mm_min_sd
_mm_max_pd
_mm_max_sd
_mm_and_pd
_mm_andnot_pd
_mm_or_pd
_mm_xor_pd
_mm_cmpeq_pd
_mm_cmplt_pd
_mm_cmple_pd
_mm_cmpgt_pd
_mm_cmpge_pd
_mm_cmpneq_pd
_mm_cmpnlt_pd
_mm_cmpnle_pd

_mm_comile_sd
_mm_comigt_sd
_mm_comige_sd
_mm_comineq_sd
_mm_ucomieq_sd
_mm_ucomilt_sd
_mm_ucomile_sd
_mm_ucomigt_sd
_mm_ucomige_sd
_mm_ucomineq_sd
_mm_load_si128
_mm_loadu_si128
_mm_loadl_epi64
_mm_store_si128
_mm_storeu_si128
_mm_storel_epi64
_mm_movepi64_pi64
_mm_move_epi64
_mm_setzero_si128
_mm_set_epi64
_mm_set_epi32
_mm_set_epi64x*
_mm_set_epi16
_mm_set_epi8
_mm_set1_epi64
_mm_set1_epi32
_mm_set1_epi64x*
_mm_set1_epi16
_mm_set1_epi8
_mm_setr_epi64
_mm_setr_epi32
_mm_setr_epi16
_mm_setr_epi8
_mm_cvtepi32_pd
_mm_cvtepi32_ps

_mm_unpacklo_pd
_mm_loadh_pd
_mm_storeh_pd
_mm_loadl_pd
_mm_storel_pd
_mm_movemask_pd
_mm_packs_epi16
_mm_packs_epi32
_mm_packus_epi16
_mm_unpackhi_epi8
_mm_unpackhi_epi16
_mm_unpackhi_epi32
_mm_unpackhi_epi64
_mm_unpacklo_epi8
_mm_unpacklo_epi16
_mm_unpacklo_epi32
_mm_unpacklo_epi64
_mm_add_epi8
_mm_add_epi16
_mm_add_epi32
_mm_add_epi64
_mm_adds_epi8
_mm_adds_epi16
_mm_adds_epu8
_mm_adds_epu16
_mm_sub_epi8
_mm_sub_epi16
_mm_sub_epi32
_mm_sub_epi64
_mm_subs_epi8
_mm_subs_epi16
_mm_subs_epu8
_mm_subs_epu16
_mm_madd_epi16
_mm_mulhi_epi16

_mm_and_si128
_mm_andnot_si128
_mm_or_si128
_mm_xor_si128
_mm_cmpeq_epi8
_mm_cmpeq_epi16
_mm_cmpeq_epi32
_mm_cmplt_epi8
_mm_cmplt_epi16
_mm_cmplt_epi32
_mm_cmpgt_epi8
_mm_cmpgt_epi16
_mm_srl_epi16
_mm_cmpgt_epi32
_mm_max_epi16
_mm_max_epu8
_mm_min_epi16
_mm_min_epu8
_mm_movemask_epi8
_mm_mulhi_epu16
_mm_maskmoveu_si128
_mm_avg_epu8
_mm_avg_epu16
_mm_sad_epu8
_mm_stream_si32
_mm_stream_si128
_mm_stream_pd
_mm_movpi64_epi64
_mm_lfence
_mm_mfence
_mm_cvtsi32_si128
_mm_cvtsi64x_si128*
_mm_cvtsi128_si32
_mm_cvtsi128_si64x*
_mm_srli_si128

_mm_cmpngt_pd	_mm_cvtpd_epi32	_mm_mullo_epi16	_mm_slli_si128
_mm_cmpnge_pd	_mm_cvtpd_pi32	_mm_mul_su32	_mm_shuffle_pd
_mm_cmpord_pd	_mm_cvtpd_ps	_mm_mul_epu32	_mm_shufflehi_epi16
_mm_cmpunord_pd	_mm_cvttpd_epi32	_mm_sll_epi16	_mm_shufflelo_epi16
_mm_cmpeq_sd	_mm_cvttpd_pi32	_mm_sll_epi32	_mm_shuffle_epi32
_mm_cmplt_sd	_mm_cvtpi32_pd	_mm_sll_epi64	_mm_extract_epi16
_mm_cmple_sd	_mm_cvtps_epi32	_mm_sra_epi16	_mm_insert_epi16
_mm_cmpgt_sd	_mm_cvttps_epi32	_mm_sra_epi32	

Table 24.4 lists the SSE3 intrinsics that PGI supports and that are available in pmmintrin.h.

Table 24.4. SSE3 Intrinsics (pmmintrin.h)

_mm_addsub_ps	_mm_moveldup_ps	_mm_loaddup_pd	_mm_mwait
_mm_hadd_ps	_mm_addsub_pd	_mm_movedup_pd	
_mm_hsub_ps	_mm_hadd_pd	_mm_lddqu_si128	
_mm_movehdup_ps	_mm_hsub_pd	_mm_monitor	

Table 24.5 lists the SSSE3 intrinsics that PGI supports and that are available in tmmintrin.h.

Table 24.5. SSSE3 Intrinsics (tmmintrin.h)

_mm_hadd_epi16	_mm_hsubs_pi16	_mm_sign_pi16
_mm_hadd_epi32	_mm_maddubs_epi16	_mm_sign_pi32
_mm_hadds_epi16	_mm_maddubs_pi16	_mm_alignr_epi8
_mm_hadd_pi16	_mm_mulhrs_epi16	_mm_alignr_pi8
_mm_hadd_pi32	_mm_mulhrs_pi16	_mm_abs_epi8
_mm_hadds_pi16	_mm_shuffle_epi8	_mm_abs_epi16
_mm_hsub_epi16	_mm_shuffle_pi8	_mm_abs_epi32
_mm_hsub_epi32	_mm_sign_epi8	_mm_abs_pi8
_mm_hsubs_epi16	_mm_sign_epi16	_mm_abs_pi16
_mm_hsub_pi16	_mm_sign_epi32	_mm_abs_pi32
_mm_hsub_pi32	_mm_sign_pi8	

Table 24.6 lists the SSE4a intrinsics that PGI supports and that are available in ammintrin.h.

Table 24.6. SSE4a Intrinsics (ammintrin.h)

_mm_stream_sd	_mm_extract_si64	_mm_insert_si64
_mm_stream_ss	_mm_extracti_si64	_mm_inserti_si64

ABM Intrinsics

PGI supports a set of ABM Intrinsics which allow the use of the ABM instructions directly from C/C++ code, without writing the assembly instructions. The following table lists the ABM intrinsics that PGI supports.

Table 24.7. ABM Intrinsics (intrin.h)

__lzcnt16	__lzcnt64	__popcnt	__rdtscp
__lzcnt	__popcnt16	__popcnt64	

Chapter 25. Messages

This chapter describes the various messages that the compiler produces. These messages include the sign-on message and diagnostic messages for remarks, warnings, and errors. The compiler always displays any error messages, along with the erroneous source line, on the screen. If you specify the –Mlist option, the compiler places any error messages in the listing file. You can also use the `-v` option to display more information about the compiler, assembler, and linker invocations and about the host system. For more information on the `-Mlist` and `-v` options, refer to Chapter 2, "*Using Command Line Options*".

Diagnostic Messages

Diagnostic messages provide syntactic and semantic information about your source text. Syntactic information includes information such as syntax errors. Semantic information includes information such as unreachable code.

You can specify that the compiler displays error messages at a certain level with the *-Minform* option.

The compiler messages refer to a severity level, a message number, and the line number where the error occurs.

The compiler can also display internal error messages on standard error. If your compilation produces any internal errors, contact The Portland Group's technical reporting service by sending e-mail to *trs@pgroup.com*.

If you use the listing file option `-Mlist`, the compiler places diagnostic messages after the source lines in the listing file, in the following format:

```
PGFTN-etype-enum-message (filename: line)
```

Where:

etype
: is a character signifying the severity level

enum
: is the error number

message
is the error message

filename
is the source filename

line
is the line number where the compiler detected an error.

Phase Invocation Messages

You can display compiler, assembler, and linker phase invocations by using the -v command line option. For further information about this option, see Chapter 2, "*Using Command Line Options*".

Fortran Compiler Error Messages

This section presents the error messages generated by the PGF77 and PGF95 compilers. The compilers display error messages in the program listing and on standard output. They can also display internal error messages on standard error.

Message Format

Each message is numbered. Each message also lists the line and column number where the error occurs. A dollar sign ($) in a message represents information that is specific to each occurrence of the message.

Message List

Error message severities:

I
informative

W
warning

S
severe error

F
fatal error

V
variable

V000 Internal compiler error. $ $
This message indicates an error in the compiler, rather than a user error – although it may be possible for a user error to cause an internal error. The severity may vary; if it is informative or warning, correct object code was probably generated, but it is not safe to rely on this. Regardless of the severity or cause, internal errors should be reported to trs@pgroup.com.

F001 Source input file name not specified
On the command line, source file name should be specified either before all the switches, or after them.

F002 Unable to open source input file: $
Source file name is misspelled, file is not in current working directory, or file is read protected.

F003 Unable to open listing file
Probably, user does not have write permission for the current working directory.

F004 $ $
Generic message for file errors.

F005 Unable to open temporary file
Compiler uses directory "/usr/tmp" or "/tmp" in which to create temporary files. If neither of these directories is available on the node on which the compiler is being used, this error will occur.

S006 Input file empty
Source input file does not contain any Fortran statements other than comments or compiler directives.

F007 Subprogram too large to compile at this optimization level $
Internal compiler data structure overflow, working storage exhausted, or some other non-recoverable problem related to the size of the subprogram. If this error occurs at opt 2, reducing the opt level to 1 may work around the problem. Moving the subprogram being compiled to its own source file may eliminate the problem. If this error occurs while compiling a subprogram of fewer than 2000 statements it should be reported to the compiler maintenance group as a possible compiler problem.

F008 Error limit exceeded
The compiler gives up because too many severe errors were issued; the error limit can be reset on the command line.

F009 Unable to open assembly file
Probably, user does not have write permission for the current working directory.

F010 File write error occurred $
Probably, file system is full.

S011 Unrecognized command line switch: $
Refer to PDS reference document for list of allowed compiler switches.

S012 Value required for command line switch: $
Certain switches require an immediately following value, such as "-opt 2".

S013 Unrecognized value specified for command line switch: $

S014 Ambiguous command line switch: $
Too short an abbreviation was used for one of the switches.

W015 Hexadecimal or octal constant truncated to fit data type

I016 Identifier, $, truncated to 31 chars
An identifier may be at most 31 characters in length; characters after the 31st are ignored.

S017 Unable to open include file: $
File is missing, read protected, or maximum include depth (10) exceeded. Remember that the file name should be enclosed in quotes.

S018 Illegal label $ $
Used for label 'field' errors or illegal values. E.g., in fixed source form, the label field (first five characters) of the indicated line contains a non-numeric character.

S019 Illegally placed continuation line
A continuation line does not follow an initial line, or more than 99 continuation lines were specified.

S020 Unrecognized compiler directive
Refer to user's manual for list of allowed compiler directives.

S021 Label field of continuation line is not blank
The first five characters of a continuation line must be blank.

S022 Unexpected end of file - missing END statement

S023 Syntax error - unbalanced $
Unbalanced parentheses or brackets.

W024 CHARACTER or Hollerith constant truncated to fit data type
A character or hollerith constant was converted to a data type that was not large enough to contain all of the characters in the constant. This type conversion occurs when the constant is used in an arithmetic expression or is assigned to a non-character variable. The character or hollerith constant is truncated on the right, that is, if 4 characters are needed then the first 4 are used and the remaining characters are discarded.

W025 Illegal character ($) - ignored
The current line contains a character, possibly non-printing, which is not a legal Fortran character (characters inside of character or Hollerith constants cannot cause this error). As a general rule, all non-printing characters are treated as white space characters (blanks and tabs); no error message is generated when this occurs. If for some reason, a non-printing character is not treated as a white space character, its hex representation is printed in the form dd where each d is a hex digit.

S026 Unmatched quote

S027 Illegal integer constant: $
Integer constant is too large for 32 bit word.

S028 Illegal real or double precision constant: $

S029 Illegal $ constant: $
Illegal hexadecimal, octal, or binary constant. A hexadecimal constant consists of digits 0..9 and letters A..F or a..f; any other character in a hexadecimal constant is illegal. An octal constant consists of digits 0..7; any other digit or character in an octal constant is illegal. A binary constant consists of digits 0 or 1; any other digit or character in a binary constant is illegal.

S030 Explicit shape must be specified for $

S031 Illegal data type length specifier for $
The data type length specifier (e.g. 4 in INTEGER*4) is not a constant expression that is a member of the set of allowed values for this particular data type.

W032 Data type length specifier not allowed for $
The data type length specifier (e.g. 4 in INTEGER*4) is not allowed in the given syntax (e.g. DIMENSION A(10)*4).

S033 Illegal use of constant $

A constant was used in an illegal context, such as on the left side of an assignment statement or as the target of a data initialization statement.

S034 Syntax error at or near $

I035 Predefined intrinsic $ loses intrinsic property

An intrinsic name was used in a manner inconsistent with the language definition for that intrinsic. The compiler, based on the context, will treat the name as a variable or an external function.

S036 Illegal implicit character range

First character must alphabetically precede second.

S037 Contradictory data type specified for $

The indicated identifier appears in more than one type specification statement and different data types are specified for it.

S038 Symbol, $, has not been explicitly declared

The indicated identifier must be declared in a type statement; this is required when the IMPLICIT NONE statement occurs in the subprogram.

W039 Symbol, $, appears illegally in a SAVE statement $

An identifier appearing in a SAVE statement must be a local variable or array.

S040 Illegal common variable $

Indicated identifier is a dummy variable, is already in a common block, or has previously been defined to be something other than a variable or array.

W041 Illegal use of dummy argument $

This error can occur in several situations. It can occur if dummy arguments were specified on a PROGRAM statement. It can also occur if a dummy argument name occurs in a DATA, COMMON, SAVE, or EQUIVALENCE statement. A program statement must have an empty argument list.

S042 $ is a duplicate dummy argument

S043 Illegal attempt to redefine $ $

An attempt was made to define a symbol in a manner inconsistent with an earlier definition of the same symbol. This can happen for a number of reasons. The message attempts to indicate the situation that occurred.

intrinsic - An attempt was made to redefine an intrinsic function. A symbol that represents an intrinsic function may be redefined if that symbol has not been previously verified to be an intrinsic function. For example, the intrinsic `sin` can be defined to be an integer array. If a symbol is verified to be an intrinsic function via the INTRINSIC statement or via an intrinsic function reference then it must be referred to as an intrinsic function for the remainder of the program unit.

symbol - An attempt was made to redefine a symbol that was previously defined. An example of this is to declare a symbol to be a PARAMETER which was previously declared to be a subprogram argument.

S044 Multiple declaration for symbol $

A redundant declaration of a symbol has occurred. For example, an attempt was made to declare a symbol as an ENTRY when that symbol was previously declared as an ENTRY.

S045 Data type of entry point $ disagrees with function $
The current function has entry points with data types inconsistent with the data type of the current function. For example, the function returns type character and an entry point returns type complex.

S046 Data type length specifier in wrong position
The CHARACTER data type specifier has a different position for the length specifier from the other data types. Suppose, we want to declare arrays ARRAYA and ARRAYB to have 8 elements each having an element length of 4 bytes. The difference is that ARRAYA is character and ARRAYB is integer. The declarations would be CHARACTER ARRAYA(8)*4 and INTEGER ARRAYB*4(8).

S047 More than seven dimensions specified for array

S048 Illegal use of '*' in declaration of array $
An asterisk may be used only as the upper bound of the last dimension.

S049 Illegal use of '*' in non-subroutine subprogram
The alternate return specifier '*' is legal only in the subroutine statement. Programs, functions, and block data are not allowed to have alternate return specifiers.

S050 Assumed size array, $, is not a dummy argument

S051 Unrecognized built-in % function
The allowable built-in functions are %VAL, %REF, %LOC, and %FILL. One was encountered that did not match one of these allowed forms.

S052 Illegal argument to %VAL or %LOC

S053 %REF or %VAL not legal in this context
The built-in functions %REF and %VAL can only be used as actual parameters in procedure calls.

W054 Implicit character $ used in a previous implicit statement
An implicit character has been given an implied data type more than once. The implied data type for the implicit character is changed anyway.

W055 Multiple implicit none statements
The IMPLICIT NONE statement can occur only once in a subprogram.

W056 Implicit type declaration
The -Mdclchk switch and an implicit declaration following an IMPLICIT NONE statement will produce a warning message for IMPLICIT statements.

S057 Illegal equivalence of dummy variable, $
Dummy arguments may not appear in EQUIVALENCE statements.

S058 Equivalenced variables $ and $ not in same common block
A common block variable must not be equivalenced with a variable in another common block.

S059 Conflicting equivalence between $ and $
The indicated equivalence implies a storage layout inconsistent with other equivalences.

S060 Illegal equivalence of structure variable, $
STRUCTURE and UNION variables may not appear in EQUIVALENCE statements.

S061 Equivalence of $ and $ extends common block backwards

W062 Equivalence forces $ to be unaligned

EQUIVALENCE statements have defined an address for the variable which has an alignment not optimal for variables of its data type. This can occur when INTEGER and CHARACTER data are equivalenced, for instance.

I063 Gap in common block $ before $

S064 Illegal use of $ in DATA statement implied DO loop

The indicated variable is referenced where it is not an active implied DO index variable.

S065 Repeat factor less than zero

S066 Too few data constants in initialization statement

S067 Too many data constants in initialization statement

S068 Numeric initializer for CHARACTER $ out of range 0 through 255

A CHARACTER*1 variable or character array element can be initialized to an integer, octal, or hexadecimal constant if that constant is in the range 0 through 255.

S069 Illegal implied DO expression

The only operations allowed within an implied DO expression are integer +, -, *, and /.

S070 Incorrect sequence of statements $

The statement order is incorrect. For instance, an IMPLICIT NONE statement must precede a specification statement which in turn must precede an executable statement.

S071 Executable statements not allowed in block data

S072 Assignment operation illegal to $ $

The destination of an assignment operation must be a variable, array reference, or vector reference. The assignment operation may be by way of an assignment statement, a data statement, or the index variable of an implied DO-loop. The compiler has determined that the identifier used as the destination is not a storage location. The error message attempts to indicate the type of entity used.

entry point - An assignment to an entry point that was not a function procedure was attempted.

external procedure - An assignment to an external procedure or a Fortran intrinsic name was attempted. If the identifier is the name of an entry point that is not a function, an external procedure.

S073 Intrinsic or predeclared, $, cannot be passed as an argument

S074 Illegal number or type of arguments to $ $

The indicated symbol is an intrinsic or generic function, or a predeclared subroutine or function, requiring a certain number of arguments of a fixed data type.

S075 Subscript, substring, or argument illegal in this context for $

This can happen if you try to doubly index an array such as ra(2)(3). This also applies to substring and function references.

S076 Subscripts specified for non-array variable $

S077 Subscripts omitted from array $

S078 Wrong number of subscripts specified for $

S079 Keyword form of argument illegal in this context for $$

S080 Subscript for array $ is out of bounds

S081 Illegal selector $ $

S082 Illegal substring expression for variable $
Substring expressions must be of type integer and if constant must be greater than zero.

S083 Vector expression used where scalar expression required
A vector expression was used in an illegal context. For example, `iscalar = iarray`, where a scalar is assigned the value of an array. Also, character and record references are not vectorizable.

S084 Illegal use of symbol $ $
This message is used for many different errors.

S085 Incorrect number of arguments to statement function $

S086 Dummy argument to statement function must be a variable

S087 Non-constant expression where constant expression required

S088 Recursive subroutine or function call of $
A function may not call itself.

S089 Illegal use of symbol, $, with character length = *
Symbols of type CHARACTER*(*) must be dummy variables and must not be used as statement function dummy parameters and statement function names. Also, a dummy variable of type CHARACTER*(*) cannot be used as a function.

S090 Hollerith constant more than 4 characters
In certain contexts, Hollerith constants may not be more than 4 characters long.

S091 Constant expression of wrong data type

S092 Illegal use of variable length character expression
A character expression used as an actual argument, or in certain contexts within I/O statements, must not consist of a concatenation involving a passed length character variable.

W093 Type conversion of expression performed
An expression of some data type appears in a context which requires an expression of some other data type. The compiler generates code to convert the expression into the required type.

S094 Variable $ is of wrong data type $
The indicated variable is used in a context which requires a variable of some other data type.

S095 Expression has wrong data type
An expression of some data type appears in a context which requires an expression of some other data type.

S096 Illegal complex comparison
The relations .LT., .GT., .GE., and .LE. are not allowed for complex values.

S097 Statement label $ has been defined more than once
More than one statement with the indicated statement number occurs in the subprogram.

S098 Divide by zero

S099 Illegal use of $
Aggregate record references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms. They may not appear, for example, in expressions. Also, records with differing structure types may not be assigned to one another.

S100 Expression cannot be promoted to a vector
An expression was used that required a scalar quantity to be promoted to a vector illegally. For example, the assignment of a character constant string to a character array. Records, too, cannot be promoted to vectors.

S101 Vector operation not allowed on $
Record and character typed entities may only be referenced as scalar quantities.

S102 Arithmetic IF expression has wrong data type
The parenthetical expression of an arithmetic if statement must be an integer, real, or double precision scalar expression.

S103 Type conversion of subscript expression for $
The data type of a subscript expression must be integer. If it is not, it is converted.

S104 Illegal control structure $
This message is issued for a number of errors involving IF-THEN statements and DO loops. If the line number specified is the last line (END statement) of the subprogram, the error is probably an unterminated DO loop or IF-THEN statement.

S105 Unmatched ELSEIF, ELSE or ENDIF statement
An ELSEIF, ELSE, or ENDIF statement cannot be matched with a preceding IF-THEN statement.

S106 DO index variable must be a scalar variable
The DO index variable cannot be an array name, a subscripted variable, a PARAMETER name, a function name, a structure name, etc.

S107 Illegal assigned goto variable $

S108 Illegal variable, $, in NAMELIST group $
A NAMELIST group can only consist of arrays and scalars which are not dummy arguments and pointer-based variables.

I109 Overflow in $ constant $, constant truncated at left
A non-decimal (hexadecimal, octal, or binary) constant requiring more than 64-bits produces an overflow. The constant is truncated at left (e.g. '1234567890abcdef1'x will be '234567890abcdef1'x).

I110 <reserved message number>

I111 Underflow of real or double precision constant

I112 Overflow of real or double precision constant

S113 Label $ is referenced but never defined

S114 Cannot initialize $

W115 Assignment to DO variable $ in loop

S116 Illegal use of pointer-based variable $ $

S117 Statement not allowed within a $ definition
The statement may not appear in a STRUCTURE or derived type definition.

S118 Statement not allowed in DO, IF, or WHERE block

I119 Redundant specification for $
Data type of indicated symbol specified more than once.

I120 Label $ is defined but never referenced

I121 Operation requires logical or integer data types
An operation in an expression was attempted on data having a data type incompatible with the operation. For example, a logical expression can consist of only logical elements of type integer or logical. Real data would be invalid.

I122 Character string truncated
Character string or Hollerith constant appearing in a DATA statement or PARAMETER statement has been truncated to fit the declared size of the corresponding identifier.

W123 Hollerith length specification too big, reduced
The length specifier field of a hollerith constant specified more characters than were present in the character field of the hollerith constant. The length specifier was reduced to agree with the number of characters present.

S124 Relational expression mixes character with numeric data
A relational expression is used to compare two arithmetic expressions or two character expressions. A character expression cannot be compared to an arithmetic expression.

I125 Dummy procedure $ not declared EXTERNAL
A dummy argument which is not declared in an EXTERNAL statement is used as the subprogram name in a CALL statement, or is called as a function, and is therefore assumed to be a dummy procedure. This message can result from a failure to declare a dummy array.

I126 Name $ is not an intrinsic function

I127 Optimization level for $ changed to opt 1 $

W128 Integer constant truncated to fit data type: $
An integer constant will be truncated when assigned to data types smaller than 32-bits, such as a BYTE.

I129 Floating point overflow. Check constants and constant expressions

I130 Floating point underflow. Check constants and constant expressions

I131 Integer overflow. Check floating point expressions cast to integer

I132 Floating pt. invalid oprnd. Check constants and constant expressions

I133 Divide by 0.0. Check constants and constant expressions

S134 Illegal attribute $ $

W135 Missing STRUCTURE name field
A STRUCTURE name field is required on the outermost structure.

W136 Field-namelist not allowed
The field-namelist field of the STRUCTURE statement is disallowed on the outermost structure.

W137 Field-namelist is required in nested structures

W138 Multiply defined STRUCTURE member name $
A member name was used more than once within a structure.

W139 Structure $ in RECORD statement not defined
A RECORD statement contains a reference to a STRUCTURE that has not yet been defined.

S140 Variable $ is not a RECORD

S141 RECORD required on left of $

S142 $ is not a member of this RECORD

S143 $ requires initializer

W144 NEED ERROR MESSAGE $ $
This is used as a temporary message for compiler development.

W145 %FILL only valid within STRUCTURE block
The %FILL special name was used outside of a STRUCTURE multiline statement. It is only valid when used within a STRUCTURE multiline statement even though it is ignored.

S146 Expression must be character type

S147 Character expression not allowed in this context

S148 Reference to $ required
An aggregate reference to a record was expected during statement compilation but another data type was found instead.

S149 Record where arithmetic value required
An aggregate record reference was encountered when an arithmetic expression was expected.

S150 Structure, Record, derived type, or member $ not allowed in this context
A structure, record, or member reference was found in a context which is not supported. For example, the use of structures, records, or members within a data statement is disallowed.

S151 Empty TYPE, STRUCTURE, UNION, or MAP
TYPE - ENDTYPE, STRUCTURE - ENDSTRUCTURE, UNION - ENDUNION MAP - ENDMAP declaration contains no members.

S152 All dimension specifiers must be ':'

S153 Array objects are not conformable $

S154 DISTRIBUTE target, $, must be a processor

S155 $ $

S156 Number of colons and triplets must be equal in ALIGN $ with $

S157 Illegal subscript use of ALIGN dummy $ - $

S158 Alternate return not specified in SUBROUTINE or ENTRY
An alternate return can only be used if alternate return specifiers appeared in the SUBROUTINE or ENTRY statements.

S159 Alternate return illegal in FUNCTION subprogram
An alternate return cannot be used in a FUNCTION.

S160 ENDSTRUCTURE, ENDUNION, or ENDMAP does not match top

S161 Vector subscript must be rank-one array

W162 Not equal test of loop control variable $ replaced with < or > test.

S163 <reserved message number>

S164 Overlapping data initializations of $
An attempt was made to data initialize a variable or array element already initialized.

S165 $ appeared more than once as a subprogram
A subprogram name appeared more than once in the source file. The message is applicable only when an assembly file is the output of the compiler.

S166 $ cannot be a common block and a subprogram
A name appeared as a common block name and a subprogram name. The message is applicable only when an assembly file is the output of the compiler.

I167 Inconsistent size of common block $
A common block occurs in more than one subprogram of a source file and its size is not identical. The maximum size is chosen. The message is applicable only when an assembly file is the output of the compiler.

S168 Incompatible size of common block $
A common block occurs in more than one subprogram of a source file and is initialized in one subprogram. Its initialized size was found to be less than its size in the other subprogram(s). The message is applicable only when an assembly file is the output of the compiler.

W169 Multiple data initializations of common block $
A common block is initialized in more than one subprogram of a source file. Only the first set of initializations apply. The message is applicable only when an assembly file is the output of the compiler.

W170 PGI Fortran extension: $ $
Use of a nonstandard feature. A description of the feature is provided.

W171 PGI Fortran extension: nonstandard statement type $

W172 PGI Fortran extension: numeric initialization of CHARACTER $

A CHARACTER*1 variable or array element was initialized with a numeric value.

W173 PGI Fortran extension: nonstandard use of data type length specifier

W174 PGI Fortran extension: type declaration contains data initialization

W175 PGI Fortran extension: IMPLICIT range contains nonalpha characters

W176 PGI Fortran extension: nonstandard operator $

W177 PGI Fortran extension: nonstandard use of keyword argument $

W178 <reserved message number>

W179 PGI Fortran extension: use of structure field reference $

W180 PGI Fortran extension: nonstandard form of constant

W181 PGI Fortran extension: & alternate return

W182 PGI Fortran extension: mixed non-character and character elements in COMMON $

W183 PGI Fortran extension: mixed non-character and character EQUIVALENCE ($,$)

W184 Mixed type elements (numeric and/or character types) in COMMON $

W185 Mixed numeric and/or character type EQUIVALENCE ($,$)

S186 Argument missing for formal argument $

S187 Too many arguments specified for $

S188 Argument number $ to $: type mismatch

S189 Argument number $ to $: association of scalar actual argument to array dummy argument

S190 Argument number $ to $: non-conformable arrays

S191 Argument number $ to $ cannot be an assumed-size array

S192 Argument number $ to $ must be a label

W193 Argument number $ to $ does not match INTENT (OUT)

W194 INTENT(IN) argument cannot be defined - $

S195 Statement may not appear in an INTERFACE block $

S196 Deferred-shape specifiers are required for $

S197 Invalid qualifier or qualifier value (/$) in OPTIONS statement
An illegal qualifier was found or a value was specified for a qualifier which does not expect a value. In either case, the qualifier for which the error occurred is indicated in the error message.

S198 $ $ in ALLOCATE/DEALLOCATE

W199 Unaligned memory reference
A memory reference occurred whose address does not meet its data alignment requirement.

S200 Missing UNIT/FILE specifier

S201 Illegal I/O specifier - $

S202 Repeated I/O specifier - $

S203 FORMAT statement has no label

S204 $ $
Miscellaneous I/O error.

S205 Illegal specification of scale factor
The integer following + or - has been omitted, or P does not follow the integer value.

S206 Repeat count is zero

S207 Integer constant expected in edit descriptor

S208 Period expected in edit descriptor

S209 Illegal edit descriptor

S210 Exponent width not used in the Ew.dEe or Gw.dEe edit descriptors

S211 Internal I/O not allowed in this I/O statement

S212 Illegal NAMELIST I/O
Namelist I/O cannot be performed with internal, unformatted, formatted, and list-directed I/O. Also, I/O lists must not be present.

S213 $ is not a NAMELIST group name

S214 Input item is not a variable reference

S215 Assumed sized array name cannot be used as an I/O item or specifier
An assumed size array was used as an item to be read or written or as an I/O specifier (i.e., FMT = array-name). In these contexts the size of the array must be known.

S216 STRUCTURE/UNION cannot be used as an I/O item

S217 ENCODE/DECODE buffer must be a variable, array, or array element

S218 Statement labeled $ $

S219 <reserved message number>

S220 Redefining predefined macro $

S221 #elif after #else
A preprocessor #elif directive was found after a #else directive; only #endif is allowed in this context.

S222 #else after #else
A preprocessor #else directive was found after a #else directive; only #endif is allowed in this context.

S223 #if-directives too deeply nested
Preprocessor #if directive nesting exceeded the maximum allowed (currently 10).

S224 Actual parameters too long for $
The total length of the parameters in a macro call to the indicated macro exceeded the maximum allowed (currently 2048).

W225 Argument mismatch for $
The number of arguments supplied in the call to the indicated macro did not agree with the number of parameters in the macro's definition.

F226 Can't find include file $
The indicated include file could not be opened.

S227 Definition too long for $
The length of the macro definition of the indicated macro exceeded the maximum allowed (currently 2048).

S228 EOF in comment
The end of a file was encountered while processing a comment.

S229 EOF in macro call to $
The end of a file was encountered while processing a call to the indicated macro.

S230 EOF in string
The end of a file was encountered while processing a quoted string.

S231 Formal parameters too long for $
The total length of the parameters in the definition of the indicated macro exceeded the maximum allowed (currently 2048).

S232 Identifier too long
The length of an identifier exceeded the maximum allowed (currently 2048).

S233 <reserved message number>

W234 Illegal directive name
The sequence of characters following a # sign was not an identifier.

W235 Illegal macro name
A macro name was not an identifier.

S236 Illegal number $
The indicated number contained a syntax error.

F237 Line too long
The input source line length exceeded the maximum allowed (currently 2048).

W238 Missing #endif
End of file was encountered before a required #endif directive was found.

W239 Missing argument list for $
A call of the indicated macro had no argument list.

S240 Number too long
The length of a number exceeded the maximum allowed (currently 2048).

W241 Redefinition of symbol $
The indicated macro name was redefined.

I242 Redundant definition for symbol $
A definition for the indicated macro name was found that was the same as a previous definition.

F243 String too long
The length of a quoted string exceeded the maximum allowed (currently 2048).

S244 Syntax error in #define, formal $ not identifier
A formal parameter that was not an identifier was used in a macro definition.

W245 Syntax error in #define, missing blank after name or arglist
There was no space or tab between a macro name or argument list and the macro's definition.

S246 Syntax error in #if
A syntax error was found while parsing the expression following a #if or #elif directive.

S247 Syntax error in #include
The #include directive was not correctly formed.

W248 Syntax error in #line
A #line directive was not correctly formed.

W249 Syntax error in #module
A #module directive was not correctly formed.

W250 Syntax error in #undef
A #undef directive was not correctly formed.

W251 Token after #ifdef must be identifier
The #ifdef directive was not followed by an identifier.

W252 Token after #ifndef must be identifier
The #ifndef directive was not followed by an identifier.

S253 Too many actual parameters to $
The number of actual arguments to the indicated macro exceeded the maximum allowed (currently 31).

S254 Too many formal parameters to $
The number of formal arguments to the indicated macro exceeded the maximum allowed (currently 31).

F255 Too much pushback
The preprocessor ran out of space while processing a macro expansion. The macro may be recursive.

W256 Undefined directive $
The identifier following a # was not a directive name.

S257 EOF in #include directive
End of file was encountered while processing a #include directive.

S258 Unmatched #elif
A #elif directive was encountered with no preceding #if or #elif directive.

S259 Unmatched #else
A #else directive was encountered with no preceding #if or #elif directive.

S260 Unmatched #endif
A #endif directive was encountered with no preceding #if, #ifdef, or #ifndef directive.

S261 Include files nested too deeply
The nesting depth of #include directives exceeded the maximum (currently 20).

S262 Unterminated macro definition for $
A newline was encountered in the formal parameter list for the indicated macro.

S263 Unterminated string or character constant
A newline with no preceding backslash was found in a quoted string.

I264 Possible nested comment
The characters /* were found within a comment.

S265 <reserved message number>

S266 <reserved message number>

S267 <reserved message number>

W268 Cannot inline subprogram; common block mismatch

W269 Cannot inline subprogram; argument type mismatch
This message may be Severe if the compilation has gone too far to undo the inlining process.

F270 Missing -exlib option

W271 Can’t inline $ - wrong number of arguments

I272 Argument of inlined function not used

S273 Inline library not specified on command line (-inlib switch)

F274 Unable to access file $/TOC

S275 Unable to open file $ while extracting or inlining

F276 Assignment to constant actual parameter in inlined subprogram

I277 Inlining of function $ may result in recursion

S278 <reserved message number>

W279 Possible use of $ before definition in $
The optimizer has detected the possibility that a variable is used before it has been assigned a value. The names of the variable and the function in which the use occurred are listed. The line number, if specified, is the line number of the basic block containing the use of the variable.

W280 Syntax error in directive $
Messages 280-300 reserved for directives. handling

W281 Directive ignored - $ $

S300 Too few data constants in initialization of derived type $

S301 $ must be TEMPLATE or PROCESSOR

S302 Unmatched END$ statement

S303 END statement for $ required in an interface block

S304 EXIT/CYCLE statement must appear in a DO/DOWHILE loop$$

S305 $ cannot be named, $

S306 $ names more than one construct

S307 $ must have the construct name $

S308 DO may not terminate at an EXIT, CYCLE, RETURN, STOP, GOTO, or arithmetic IF

S309 Incorrect name, $, specified in END statement

S310 $ $
Generic message for MODULE errors.

W311 Non-replicated mapping for $ array, $, ignored

W312 Array $ should be declared SEQUENCE

W313 Subprogram $ called within INDEPENDENT loop not PURE

E314 IPA: actual argument $ is a label, but dummy argument $ is not an asterisk
The call passes a label to the subprogram; the corresponding dummy argument in the subprogram should be an asterisk to declare this as the alternate return.

I315 IPA: routine $, $ constant dummy arguments
This many dummy arguments are being replaced by constants due to interprocedural analysis.

I316 IPA: routine $, $ INTENT(IN) dummy arguments
This many dummy arguments are being marked as INTENT(IN) due to interprocedural analysis.

I317 IPA: routine $, $ array alignments propagated
This many array alignments were propagated by interprocedural analysis.

I318 IPA: routine $, $ distribution formats propagated
This many array distribution formats were propagated by interprocedural analysis.

I319 IPA: routine $, $ distribution targets propagated
This many array distribution targets were propagated by interprocedural analysis.

I320 IPA: routine $, $ common blocks optimized
This many mapped common blocks were optimized by interprocedural analysis.

I321 IPA: routine $, $ common blocks not optimized
This many mapped common blocks were not optimized by interprocedural analysis, either because they were declared differently in different routines, or they did not appear in the main program.

I322 IPA: analyzing main program $
Interprocedural analysis is building the call graph and propagating information with the named main program.

I323 IPA: collecting information for $
Interprocedural analysis is saving information for the current subprogram for subsequent analysis and propagation.

W324 IPA file $ appears to be out of date

W325 IPA file $ is for wrong subprogram: $

W326 Unable to open file $ to propagate IPA information to $

I327 IPA: $ subprograms analyzed

I328 IPA: $ dummy arguments replaced by constants

I329 IPA: $ INTENT(IN) dummy arguments should be INTENT(INOUT)

I330 IPA: $ dummy arguments changed to INTENT(IN)

I331 IPA: $ inherited array alignments replaced

I332 IPA: $ transcriptive distribution formats replaced

I333 IPA: $ transcriptive distribution targets replaced

I334 IPA: $ descriptive/prescriptive array alignments verified

I335 IPA: $ descriptive/prescriptive distribution formats verified

I336 IPA: $ descriptive/prescriptive distribution targets verified

I337 IPA: $ common blocks optimized

I338 IPA: $ common blocks not optimized

S339 Bad IPA contents file: $

S340 Bad IPA file format: $

S341 Unable to create file $ while analyzing IPA information

S342 Unable to open file $ while analyzing IPA information

S343 Unable to open IPA contents file $

S344 Unable to create file $ while collecting IPA information

F345 Internal error in $: table overflow
Analysis failed due to a table overflowing its maximum size.

W346 Subprogram $ appears twice
The subprogram appears twice in the same source file; IPA will ignore the first appearance.

F347 Missing -ipalib option
Interprocedural analysis, enabled with the `–ipacollect`, `–ipaanalyze`, or `–ipapropagate` options, requires the `-ipalib` option to specify the library directory.

W348 Common /$/ $ has different distribution target
The array was declared in a common block with a different distribution target in another subprogram.

W349 Common /$/ $ has different distribution format
The array was declared in a common block with a different distribution format in another subprogram.

W350 Common /$/ $ has different alignment
The array was declared in a common block with a different alignment in another subprogram.

W351 Wrong number of arguments passed to $
The subroutine or function statement for the given subprogram has a different number of dummy arguments than appear in the call.

W352 Wrong number of arguments passed to $ when bound to $
The subroutine or function statement for the given subprogram has a different number of dummy arguments than appear in the call to the EXTERNAL name given.

W353 Subprogram $ is missing
A call to a subroutine or function with this name appears, but it could not be found or analyzed.

I354 Subprogram $ is not called
No calls to the given subroutine or function appear anywhere in the program.

W355 Missing argument in call to $
A nonoptional argument is missing in a call to the given subprogram.

I356 Array section analysis incomplete
Interprocedural analysis for array section arguments is incomplete; some information may not be available for optimization.

I357 Expression analysis incomplete
Interprocedural analysis for expression arguments is incomplete; some information may not be available for optimization.

W358 Dummy argument $ is EXTERNAL, but actual is not subprogram
The call statement passes a scalar or array to a dummy argument that is declared EXTERNAL.

W359 SUBROUTINE $ passed to FUNCTION dummy argument $
The call statement passes a subroutine name to a dummy argument that is used as a function.

W360 FUNCTION $ passed to FUNCTION dummy argument $ with different result type
The call statement passes a function argument to a function dummy argument, but the dummy has a different result type.

W361 FUNCTION $ passed to SUBROUTINE dummy argument $
The call statement passes a function name to a dummy argument that is used as a subroutine.

W362 Argument $ has a different type than dummy argument $
The type of the actual argument is different than the type of the corresponding dummy argument.

W363 Dummy argument $ is a POINTER but actual argument $ is not
The dummy argument is a pointer, so the actual argument must be also.

W364 Array or array expression passed to scalar dummy argument $
The actual argument is an array, but the dummy argument is a scalar variable.

W365 Scalar or scalar expression passed to array dummy argument $
The actual argument is a scalar variable, but the dummy argument is an array.

F366 Internal error: interprocedural analysis fails
An internal error occurred during interprocedural analysis; please report this to the compiler maintenance group. If user errors were reported when collecting IPA information or during IPA analysis, correcting them may avoid this error.

I367 Array $ bounds cannot be matched to formal argument
Passing a nonsequential array to a sequential dummy argument may require copying the array to sequential storage. The most common cause is passing an ALLOCATABLE array or array expression to a dummy argument that is declared with explicit bounds. Declaring the dummy argument as assumed shape, with bounds (:,:,:), will remove this warning.

W368 Array-valued expression passed to scalar dummy argument $
The actual argument is an array-valued expression, but the dummy argument is a scalar variable.

W369 Dummy argument $ has different rank than actual argument
The actual argument is an array or array-valued expression with a different rank than the dummy argument.

W370 Dummy argument $ has different shape than actual argument
The actual argument is an array or array-valued expression with a different shape than the dummy argument; this may require copying the actual argument into sequential storage.

W371 Dummy argument $ is INTENT(IN) but may be modified
The dummy argument was declared as INTENT(IN), but analysis has found that the argument may be modified; the INTENT(IN) declaration should be changed.

W372 Cannot propagate alignment from $ to $
The most common cause is when passing an array with an inherited alignment to a dummy argument with non-inherited alignment.

I373 Cannot propagate distribution format from $ to $
The most common cause is when passing an array with a transcriptive distribution format to a dummy argument with prescriptive or descriptive distribution format.

I374 Cannot propagate distribution target from $ to $
The most common cause is when passing an array with a transcriptive distribution target to a dummy argument with prescriptive or descriptive distribution target.

I375 Distribution format mismatch between $ and $
Usually this arises when the actual and dummy arguments are distributed in different dimensions.

I376 Alignment stride mismatch between $ and $
This may arise when the actual argument has a different stride in its alignment to its template than does the dummy argument.

I377 Alignment offset mismatch between $ and $
This may arise when the actual argument has a different offset in its alignment to its template than does the dummy argument.

I378 Distribution target mismatch between $ and $
This may arise when the actual and dummy arguments have different distribution target sizes.

I379 Alignment of $ is too complex
The alignment specification of the array is too complex for interprocedural analysis to verify or propagate; the program will work correctly, but without the benefit of IPA.

I380 Distribution format of $ is too complex
The distribution format specification of the array is too complex for interprocedural analysis to verify or propagate; the program will work correctly, but without the benefit of IPA.

I381 Distribution target of $ is too complex
The distribution target specification of the array is too complex for interprocedural analysis to verify or propagate; the program will work correctly, but without the benefit of IPA.

I382 IPA: $ subprograms analyzed
Interprocedural analysis succeeded in finding and analyzing this many subprograms in the whole program.

I383 IPA: $ dummy arguments replaced by constants
Interprocedural analysis has found this many dummy arguments in the whole program that can be replaced by constants.

I384 IPA: $ dummy arguments changed to INTENT(IN)
Interprocedural analysis has found this many dummy arguments in the whole program that are not modified and can be declared as INTENT(IN).

W385 IPA: $ INTENT(IN) dummy arguments should be INTENT(INOUT)
Interprocedural analysis has found this many dummy arguments in the whole program that were declared as INTENT(IN) but should be INTENT(INOUT).

I386 IPA: $ array alignments propagated
Interprocedural analysis has found this many array dummy arguments that could have the inherited array alignment replaced by a descriptive alignment.

I387 IPA: $ array alignments verified
Interprocedural analysis has verified that the prescriptive or descriptive alignments of this many array dummy arguments match the alignments of the actual argument.

I388 IPA: $ array distribution formats propagated
Interprocedural analysis has found this many array dummy arguments that could have the transcriptive distribution format replaced by a descriptive format.

I389 IPA: $ array distribution formats verified
Interprocedural analysis has verified that the prescriptive or descriptive distribution formats of this many array dummy arguments match the formats of the actual argument.

I390 IPA: $ array distribution targets propagated
Interprocedural analysis has found this many array dummy arguments that could have the transcriptive distribution target replaced by a descriptive target.

I391 IPA: $ array distribution targets verified
Interprocedural analysis has verified that the prescriptive or descriptive distribution targets of this many array dummy arguments match the targets of the actual argument.

I392 IPA: $ common blocks optimized
Interprocedural analysis has found this many common blocks that could be optimized.

I393 IPA: $ common blocks not optimized
Interprocedural analysis has found this many common blocks that could not be optimized, either because the common block was not declared in the main program, or because it was declared differently in different subprograms.

I394 IPA: $ replaced by constant value
The dummy argument was replaced by a constant as per interprocedural analysis.

I395 IPA: $ changed to INTENT(IN)
The dummy argument was changed to INTENT(IN) as per interprocedural analysis.

I396 IPA: array alignment propagated to $
The template alignment for the dummy argument was changed as per interprocedural analysis.

I397 IPA: distribution format propagated to $
The distribution format for the dummy argument was changed as per interprocedural analysis.

I398 IPA: distribution target propagated to $
The distribution target for the dummy argument was changed as per interprocedural analysis.

I399 IPA: common block $ not optimized
The given common block was not optimized by interprocedural analysis either because it was not declared in the main program, or because it was declared differently in different subprograms.

E400 IPA: dummy argument $ is an asterisk, but actual argument is not a label
The subprogram expects an alternate return label for this argument.

E401 Actual argument $ is a subprogram, but Dummy argument $ is not declared EXTERNAL
The call statement passes a function or subroutine name to a dummy argument that is a scalar variable or array.

E402 Actual argument $ is illegal

E403 Actual argument $ and formal argument $ have different ranks
The actual and formal array arguments differ in rank, which is allowed only if both arrays are declared with the HPF SEQUENCE attribute.

E404 Sequential array section of $ in argument $ is not contiguous
When passing an array section to a formal argument that has the HPF SEQUENCE attribute, the actual argument must be a whole array with the HPF SEQUENCE attribute, or an array section of such an array where the section is a contiguous sequence of elements.

E405 Array expression argument $ may not be passed to sequential dummy argument $
When the dummy argument has the HPF SEQUENCE attribute, the actual argument must be a whole array with the HPF SEQUENCE attribute or a contiguous array section of such an array, unless an INTERFACE block is used.

E406 Actual argument $ and formal argument $ have different character lengths
The actual and formal array character arguments have different character lengths, which is allowed only if both character arrays are declared with the HPF SEQUENCE attribute, unless an INTERFACE block is used.

W407 Argument $ has a different character length than dummy argument $
The character length of the actual argument is different than the length specified for the corresponding dummy argument.

W408 Specified main program $ is not a PROGRAM
The main program specified on the command line is a subroutine, function, or block data subprogram.

W409 More than one main program in IPA directory: $ and $
There is more than one main program analyzed in the IPA directory shown. The first one found is used.

W410 No main program found; IPA analysis fails.
The main program must appear in the IPA directory for analysis to proceed.

W411 Formal argument $ is DYNAMIC but actual argument is an expression

W412 Formal argument $ is DYNAMIC but actual argument $ is not

I413 Formal argument $ has two reaching distributions and may be a candidate for cloning

I414 $ and $ may be aliased and one of them is assigned

Interprocedural analysis has determined that two formal arguments may be aliased because the same variable is passed in both argument positions; or one formal argument and a global or COMMON variable may be aliased, because the global or COMMON variable is passed as an actual argument. If either alias is assigned in the subroutine, unexpected results may occur; this message alerts the user that this situation is disallowed by the Fortran standard.

F415 IPA fails: incorrect IPA file

Interprocedural analysis saves its information in special IPA files in the specified IPA directory. One of these files has been renamed or corrupted. This can arise when there are two files with the same prefix, such as `a.hpf` and `a.f90`.

E416 Argument $ has the SEQUENCE attribute, but the dummy parameter $ does not

When an actual argument is an array with the SEQUENCE attribute, the dummy parameter must have the SEQUENCE attribute or an INTERFACE block must be used.

E417 Interface block for $ is a SUBROUTINE but should be a FUNCTION

E418 Interface block for $ is a FUNCTION but should be a SUBROUTINE

E419 Interface block for $ is a FUNCTION has wrong result type

W420 Earlier $ directive overrides $ directive

W421 $ directive can only appear in a function or subroutine

E422 Nonconstant DIM= argument is not supported

E423 Constant DIM= argument is out of range

E424 Equivalence using substring or vector triplets is not allowed

E425 A record is not allowed in this context

E426 WORD type cannot be converted

E427 Interface block for $ has wrong number of arguments

E428 Interface block for $ should have $

E429 Interface block for $ should not have $

E430 Interface block for $ has wrong $

W431 Program is too large for Interprocedural Analysis to complete

W432 Illegal type conversion $

E433 Subprogram $ called within INDEPENDENT loop not LOCAL

W434 Incorrect home array specification ignored

S435 Array declared with zero size
An array was declared with a zero or negative dimension bound, as 'real a(-1)', or an upper bound less than the lower bound, as 'real a(4:2)'.

W436 Independent loop not parallelized$

W437 Type $ will be mapped to $
Where DOUBLE PRECISION is not supported, it is mapped to REAL, and similarly for COMPLEX(16) or COMPLEX*32.

E438 $ $ not supported on this platform
This construct is not supported by the compiler for this target.

S439 An internal subprogram cannot be passed as argument - $

S440 Defined assignment statements may not appear in WHERE statement or WHERE block

S441 $ may not appear in a FORALL block

E442 Adjustable-length character type not supported on this host - $ $

S443 EQUIVALENCE of derived types not supported on this host - $

S444 Derived type in EQUIVALENCE statement must have SEQUENCE attribute - $
A variable or array with derived type appears in an EQUIVALENCE statement. The derived type must have the SEQUENCE attribute, but does not.

E445 Array bounds must be integer $ $
The expressions in the array bounds must be integer.

S446 Argument number $ to $: rank mismatch
The number of dimensions in the array or array expression does not match the number of dimensions in the dummy argument.

S447 Argument number $ to $ must be a subroutine or function name

S448 Argument number $ to $ must be a subroutine name

S449 Argument number $ to $ must be a function name

S450 Argument number $ to $: kind mismatch

S451 Arrays of derived type with a distributed member are not supported

S452 Assumed length character, $, is not a dummy argument

S453 Derived type variable with pointer member not allowed in IO - $ $

S454 Subprogram $ is not a module procedure
Only names of module procedures declared in this module or accessed through USE association can appear in a MODULE PROCEDURE statement.

S455 A derived type array section cannot appear with a member array section - $
A reference like A(:)%B(:), where 'A' is a derived type array and 'B' is a member array, is not allowed; a section subscript may appear after 'A' or after 'B', but not both.

S456 Unimplemented for data type for MATMUL

S457 Illegal expression in initialization

S458 Argument to NULL() must be a pointer

S459 Target of NULL() assignment must be a pointer

S460 ELEMENTAL procedures cannot be RECURSIVE

S461 Dummy arguements of ELEMENATAL procedures must be scalar

S462 Arguments and return values of ELEMENATAL procedures cannot have the POINTER attribute

S463 Arguments of ELEMENATAL procedures cannot be procedures

S464 An ELEMENTAL procedure cannot be passed as argument - $

Fortran Run-time Error Messages

This section presents the error messages generated by the run-time system. The run-time system displays error messages on standard output.

Message Format

The messages are numbered but have no severity indicators because they all terminate program execution.

Message List

Here are the run-time error messages:

201 illegal value for specifier
An improper specifier value has been passed to an I/O run-time routine. Example: within an OPEN statement, form='unknown'.

202 conflicting specifiers
Conflicting specifiers have been passed to an I/O run-time routine. Example: within an OPEN statement, form='unformatted', blank='null'.

203 record length must be specified
A recl specifier required for an I/O run-time routine has not been passed. Example: within an OPEN statement, access='direct' has been passed, but the record length has not been specified (recl=specifier).

204 illegal use of a readonly file
Self explanatory. Check file and directory modes for readonly status.

205 'SCRATCH' and 'SAVE'/'KEEP' both specified
In an OPEN statement, a file disposition conflict has occurred. Example: within an OPEN statement, status='scratch' and dispose='keep' have been passed.

206 attempt to open a named file as 'SCRATCH'

207 file is already connected to another unit

208 'NEW' specified for file that already exists

209 'OLD' specified for file that does not exist

210 dynamic memory allocation failed
Memory allocation operations occur only in conjunction with namelist I/O. The most probable cause of fixed buffer overflow is exceeding the maximum number of simultaneously open file units.

211 invalid file name

212 invalid unit number
A file unit number less than or equal to zero has been specified.

215 formatted/unformatted file conflict
Formatted/unformatted file operation conflict.

217 attempt to read past end of file

219 attempt to read/write past end of record
For direct access, the record to be read/written exceeds the specified record length.

220 write after last internal record

221 syntax error in format string
A run-time encoded format contains a lexical or syntax error.

222 unbalanced parentheses in format string

223 illegal P or T edit descriptor - value missing

224 illegal Hollerith or character string in format
An unknown token type has been found in a format encoded at run-time.

225 lexical error -- unknown token type

226 unrecognized edit descriptor letter in format
An unexpected Fortran edit descriptor (FED) was found in a run-time format item.

228 end of file reached without finding group

229 end of file reached while processing group

230 scale factor out of range -128 to 127
Fortran P edit descriptor scale factor not within range of -128 to 127.

231 error on data conversion

233 too many constants to initialize group item

234 invalid edit descriptor
An invalid edit descriptor has been found in a format statement.

235 edit descriptor does not match item type
Data types specified by I/O list item and corresponding edit descriptor conflict.

236 formatted record longer than 2000 characters

237 quad precision type unsupported

238 tab value out of range
A tab value of less than one has been specified.

239 entity name is not member of group

242 illegal operation on direct access file

243 format parentheses nesting depth too great

244 syntax error - entity name expected

245 syntax error within group definition

246 infinite format scan for edit descriptor

248 illegal subscript or substring specification

249 error in format - illegal E, F, G or D descriptor

250 error in format - number missing after '.', '-', or '+'

251 illegal character in format string

252 operation attempted after end of file

253 attempt to read non-existent record (direct access)

254 illegal repeat count in format

Index

Symbols

!DEC$ directive, 334
%eax, 341
%ebp, 341
%ecx, 341
%edi, 341
%edx, 341
%esi, 341
%esp, 341
%rax, 348
%rbp, 347, 358
%rdi, 348
%rsp, 347, 358
%st(0), 341
%st(1), 341
64-Bit Programming, 163
  compiler options, 165
  data types, 163
-Bdynamic, 121
-dryrun
  as diagnostic tool, 26
-help
  Options
    -help, 26
-Mconcur, 35, 47
  altcode option, 35
  cncall option, 35
  dist option, 35
  suboptions, 35
-Mextract
  suboptions, 51
-Minfo, 26
-Minline, 49
  suboptions, 49
-Miomutex, 60
-Mipa, 39
-Mneginfo, 26
-mp, 47, 60
-Mpfi, 44
-Mpfo, 44
-Mreentrant, 60
-Msafe_lastval, 38
-Mvect, 30, 32
-tp, 39
  using, 21

A

Accelerator
  using, 81, 311
ALIAS
  ATTRIBUTES list, 335
ALIAS directive, 334
Aliases
  operand, 183
altcode directive, 328
altcode pragma, 328
ar command, 116
Arguments
  floating point, 343, 349, 359
  integral, 343, 349, 359
  Inter-language calling, 151
  passing, 151, 353, 359
  passing by reference, 362
  passing by value, 151, 362
  passing on stack, 350
  pointer, 343, 349, 359
  structures, 343, 349, 359
  unassigned, 349
  union, 343, 349, 359
Arrays, 354
  64-bit indexing, 164
  64-bit options, 165
  indices, 152
  large, 165
Assembly
  string modifier characters, 184
Assembly Language
  called routine, 345
  called routine in C, 351
assoc directive, 329
assoc pragma, 329
ATOMIC directive, 286
atomic pragma, 286
ATTRIBUTES Directive, 335
  ALIAS, 335
  C, 335
  DLLEXPORT, 335
  DLLIMPORT, 335
  NOMIXED_STR_LEN_ARG, 335
  REFERENCE, 336
  STDCALL, 336
  VALUE, 336
Auto-parallelization, 35
  failure, 36
  sub-options, 35

B

BARRIER directive, 287
Barriers
  explicit, 284
  implicit, 284
bash shell
  initialization, 10
  instance, 10
Bdynamic, 118
BLAS library, 125
Blocks
  basic, defined, 24
  blank common, 355, 365
  common, 355, 364
  common, Fortran, 150
  Fortran named common, 150
Bounds checking, 275
bounds directive, 329
Bstatic, 118
Build
  command-line options, 195
  DLLS, 120
  DLLS containing circular mutual imports, 121
  DLLS containing mutual imports, 122
  DLLs example , 119
  program using Make, 40
  program with IPA, 41

program without IPA, 39, 40

C

C
ATTRIBUTES directive, 335
C/C++
builtin functions, 111
math header file, 111
C$PRAGMA C, 110
C++
classes, scopes, 323
name mangling, 323
parallelization pragmas, 59
pragmas, 59
Standard Template Library, 125
Cache tiling
failed cache tiling, 279
with -Mvect, 273
Calling conventions
CREF, 161
overview, 147
STDCALL, 160
UNIX, 161
Win32, 160
Calls
inter-language, 353, 363
CCP_HOME, 78
CCP_SDK, 78
Clauses
directives, 59
driectives, 62
pragmas, 62
Clobber list, 176
cncall directive, 329
cncall pragma, 329
Code
generation, 144
mutiple processors, 144
optimization, 23
parallelization, 23
position indendent, 165
processor-specific, 144
speed, 36
x86 generation, 144
Collection
IPA phase, 41
Command line
case sensitivity, 2
conflicting options rules, 18
include files, 5
option order, 3
suboptions, 18
Command-line Options, 3, 17, 195, 217
-#, 203
-###, 203
-+p, 250
-A, 242
-a, 243
-alias, 243
-B, 244
-b, 244
-b3, 244
-Bdynamic, 203
-Bstatic, 204
-Bstatic_pgi, 204
Build-related, 195
-byteswapio, 205
-C, 205
-c, 206
--cfront_2.1, 245
--cfront_3.0, 245
--compress_names, 246
--create_pch, 246
-d, 206
-D, 206
Debug-related, 198, 199, 199
--diag_error, 246
--diag_remark, 247
--diag_suppress, 247
--diag_warning, 247
--display_error_number, 247
-dryrun, 207, 208
-dynamiclib, 208
-E, 208
-e, 248
-F, 208
-fast, 209
-fastsse, 209
-flagcheck, 209
-flags, 210
-fpic, 210
-fPIC, 210
-G, 210
-g, 211
-g77libs, 211
Generic PGI options, 203
--gnu_extensions, 248
-gopt, 211
help, 18
-help, 212
-I, 213
-i2, -i4 and -i8, 214
--keeplnk, 216
-Kflag, 214
-L, 216
-l, 217
-m, 217
-M, 249
makefiles, 18
-Mallocatable, 261
-Manno, 275
-Masmkeyword, 259
-Mbackslash, 261
-Mbounds, 275
-Mbyteswapio, 275
-Mcache_align, 265
-Mchkfpstk, 275
-Mchkptr, 276
-Mchkstk, 276
-mcmodel=medium, 164
-mcmodel=small, 164
-Mconcur, 265
-Mcpp, 276
-Mcray, 266
-MD, 249
-Mdaz, 254
-Mdclchk, 261
-Mdefaultunit, 261
-Mdepchk, 266
-Mdlines, 261
-Mdll, 277
-Mdollar, 259, 262
-Mdse, 266
-Mdwarf1, 254, 254
-Mdwarf2, 254
-Mdwarf3, 254
-Mextend, 262

-Mextract, 263
-Mfcon, 259
-Mfixed, 262
-Mflushz, 254
-Mfpapprox, 266
-Mfpmisalign, 267
-Mfprelaxed, 267
-Mfree, 262
-Mfunc32, 254
-Mgccbugs, 277, 277
-Mi4, 267
-Minfo, 277
-Minform, 278, 279
-Minline, 263
-Minstrument, 254
-Miomutex, 262
-Mipa, 267
-Mkeepasm, 279
-Mlarge_arrays, 164, 255
-Mlargeaddressaware, 255
-Mlfs, 260
-Mlist, 279
-Mloop32, 269
-Mlre, 270
-Mm128, 259
-Mmakedll, 279
-Mmakeimplib, 279
-Mnames, 279
-Mneginfo, 279
-Mnoasmkeyword, 259
-Mnobackslash, 261
-Mnobounds, 275
-Mnodaz, 254
-Mnodclchk, 261
-Mnodefaultunit, 261
-Mnodepchk, 266
-Mnodlines, 262
-Mnodse, 266
-Mnoflushz, 254
-Mnofpapprox, 266
-Mnofpmisalign, 267
-Mnofprelaxed, 267, 267
-Mnoframe, 270
-Mnoi4, 270
-Mnoiomutex, 262
-Mnolarge_arrays, 255, 256
-Mnolist, 280
-Mnoloop32, 269
-Mnolre, 270
-Mnom128, 259
-Mnomain, 256
-Mnoonetrip, 262
-Mnoopenmp, 281
-Mnopgdllmain, 281
-Mnoprefetch, 271
-Mnor8, 271
-Mnor8intrinsics, 271
-Mnorecursive, 257
-Mnoreentrant, 257
-Mnoref_externals, 257
-Mnorpath, 281
-Mnosave, 262
-Mnoscalarsse, 272
-Mnosecond_underscore, 257
-Mnosgimp, 281
-Mnosignextend, 257
-Mnosingle, 259
-Mnosmart, 272
-Mnostartup, 260
-Mnostddef, 261
-Mnostdlib, 261, 261
-Mnostride0, 258
-Mnounixlogical, 262
-Mnounroll, 273
-Mnoupcase, 263
-Mnovect, 274
-Mnovintr, 274, 274
-module, 224
-Monetrip, 262
-mp, 224
-Mpfi, 270
-Mpfo, 270, 270
-Mpre, 256
-Mprefetch, 271
-Mpreprocess, 281
-Mprof, 256
-Mr8, 271
-Mr8intrinsics, 271
-Mrecursive, 257
-Mreentrant, 257
-Mref_externals, 257
-Msafe_lastval, 258
-Msafeptr, 271
-Msave, 262
-Mscalarsse, 272
-Mschar, 259
-Msecond_underscore, 257
-Msignextend, 257
-Msingle, 259
-Msmart, 272
-Msmartalloc, 260
-Mstandard, 262
-Mstride0, 258
-Muchar, 259
-Munix, 258
-Munixlogical, 262
-Munroll, 272
-Mupcase, 262
-Mvarargs, 258
-Mvect, 273
-Mwritable_strings, 281
--llalign, 248, 256, 263, 271, 272
--alternative_tokens, 243, 245, 248, 251, 252
nontemporal move, 256
-noswitcherror, 225
-O, 226
-o, 227
--optk_allow_dollar_in_id_chars, 249
-P, 249
-pc, 227
--pch, 250
--pch_dir, 250
--pedantic, 229
-pg, 229
-pgf77libs, 230, 230
-pgf90libs, 230
--preinclude, 251
-R, 230
-r, 231
-r4 and -r8, 231
-rc, 231
redundancy elimination, 256
-rpath, 232
rules of use, 3
-s, 232

-S, 232
-shared, 233
-show, 233
-silent, 233
-soname, 234
-stack, 234
suboptions, 18
syntax, 2, 17
-t, 252
-time, 235, 236
-tp, 236
-u, 239
-U, 240
--use_pch, 251
-V, 240
-v, 240
-W, 241
-w, 241
-X, 252
-Xs, 242
-Xt, 242
--zc_eh, 253
Commands
ar, 116
dir, 52
ls, 52
ranlib, 117
Compilation driver, 1
Compiler
64-bit options, 165
inform, 327
Compiler options
64-bit, 165
effects on memory, array sizes, 166
Compilers
Invoke at command level, 1
PGC++, xxv
PGF77, xxv
PGF95, xxv
PGHPF, xxv
concur directive, 329
concur pragma, 329
Constants
logical, 352
Constraints
*, 182, 182
&, 182
%, 182
+, 182
=, 182
character, 177
inline assembly, 177
machine, 179, 179
machine, example, 180
modifiers, 181
multiple alternative, 181, 181
operand, 177
operand aliases, 183
simple, 177
Control word, 341
Conventions
runtime on x86 processor, 339
Count
instructions, 45
cpp, 5
CPU_CLOCK, 45
Create
inline library, 51
shared object files, 112
CREF
calling conventions, 161
CRITICAL directive, 287
Critical pragma, 287
Customization
site-specific, 13

D

Data Types, 7
Data types, 187, 353
64-bit, 163
Aggregate, 7
attributes, 194
bit-fields, 194
C/C++, 164
C/C++ aggregate alignment, 193
C/C++ scalar data types, 190
C/C++ struct, 192
C/C++ void, 194
C++ class and object layout, 192
C++ classes, 192
compatibility of Fortran and C/C++, 149
DEC structures, 189
DEC Unions, 189
F90 derived types, 190
Fortran, 164
Fortran representation, 187
Fortran scalars, 187
inter-language calling, 149
internal padding, 193
Linux large static, 164
Real ranges, 188
scalars, 7, 188
tail padding, 193
Debug
command-line options, 198
JIT, 139
DECORATE directive, 336
Default
Win32 calling conventions, 160
depchk directive, 329
depchk pragma, 329
Deployment, 141, 141
Linux 64-bit, 142
Development
common tasks, 14
Diagnostics
-dryrun, 26
dir command, 52
Directives, 101
ALIAS, 334
altcode, 328
assoc, 329
ATOMIC, 286
ATTRIBUTES, 335
BARRIER, 287
bounds, 329
C/C++, 3
clauses, 59, 62
cncall, 329
concur, 329
CRITICAL...END CRITICAL, 287
DECORATE, 336
default scopes, 102
depchk, 329
DISTRIBUTE, 336, 337

eqvchk, 329
Fortran, 3, 3
Fortran parallization overview, 59
global scopes, 101
IDEC$, 109, 334
invarif, 330
ivdep, 330
loop scopes, 101, 102
lstval, 330
-Miomutex option, 60
-mp option, 60
-Mreentrant option, 60
name, 59
noaltcode, 328
noassoc, 329
nobounds, 329
nocncall, 329
noconcur, 329
nodepchk, 329
noeqvchk, 329
noinvarif, 330
nolstval, 330
nosafe_lastval, 331
nounroll, 333
novector, 334
novintr, 334
optimization, 101, 327
Parallelization, 55, 283, 327
parallelization, 59
prefetch, 107, 330, 330, 334
prefetch example, 108
prefetch sentinel, 108
prefetch syntax, 108, 334
recognition, 60
routine scopes, 101
safe_lastval, 331
scope, 104
scope indicator, 101, 327
Summary table, 60, 102, 109
syntax, 59
tp, 333
Unified Binary, 145
unroll, 333
valid scopes, 101
vector, 334
vintr, 334
Distribute
  files, 141
DISTRIBUTE directive, 336, 337
Distributing Files, 141
DLLEXPORT
  ATTRIBUTES directive, 335
DLLIMPORT
  ATTRIBUTES directive, 335
DLLs
  Bdynamic, 118
  Bstatic, 118
  Build steps in C, 120
  Build steps in Fortran, 119
  example, 120
  generate .def file, 119
  import library, 119
  library without dll, 119
  Mmakedll, 119
  name, 119
DOACROSS directive, 289
DO directive, 290
Dynamic
  large dynamically allocated data, 164
  libraries on Mac OS X, 115
  link libraries on Windows, 118

E

EFLAGS, 341
Environment variables, 127
  directives, 70, 74
  FLEXLM_BATCH, 129, 131
  FORTRAN_OPT, 129, 131, 131, 131, 131
  GMON_OUT_PREFIX, 129, 131
  HP-MPI, MPI_ROOT, 77
  LD_LIBRARY_PATH, 113, 129, 131
  LM_LICENSE_FILE, 129, 132
  MANPATH, 129, 132
  MCPUS, 36, 129
  MP_BIND, 129, 132
  MP_BLIST, 129, 133
  MP_SPIN, 129, 133
  MP_WARN, 129, 133
  MPI_ROOT, 77
  MPI, CCP_HOME, 78
  MPI, CCP_SDK, 78
  MPI, MPIDIR, 76, 76, 77
  MPIDIR, 76, 76, 77
  MPSTKZ, 129, 132
  NCPUS, 133
  NCPUS_MAX, 129, 134
  NO_STOP_MESSAGE, 129, 134
  OMP_DYNAMIC, 130, 130
  OMP_NESTED, 130
  OMP_NUM_THREADS, 130
  OMP_STACK_SIZE, 9, 11, 13, 71, 130, 309
  OMP_THREAD_LIMIT, 310
  OMP_WAIT_POLICY, 71, 130, 310
  OpenMP, 70, 74, 308
  OpenMP, OMP_DYNAMIC, 308
  OpenMP, OMP_MAX_ACTIVE_LEVELS, 308
  OpenMP, OMP_NESTED, 308
  OpenMP, OMP_NUM_THREADS, 309
  OpenMP, OMP_SCHEDULE, 309
  OpenMP, OMP_STACK_SIZE, 309
  OpenMP, OMP_THREAD_LIMIT, 310
  OpenMP, OMP_WAIT_POLICY, 310
  OpenMP Summary Table, 70
  PATH, 130, 134
  PGI, 130, 130, 134
  PGI_CONTINUE, 130, 135
  PGI_OBJSUFFIX, 130, 135
  PGI_STACK_USAGE, 135, 276
  PGI_TERM, 130, 135
  PGI_TERM_DEBUG, 130, 130, 136, 137
  PGI-related, 129
  PWD, 137
  setting, 127
  setting on Linux, 127
  setting on Mac OS X, 128
  setting on Windows, 128
  STATIC_RANDOM_SEED, 130, 137
  TMP, 130, 138

TMPDIR, 130, 138
using, 138
eqvchk directive, 329
eqvchk pragma, 329
Errors
inlining, 53
Examples
Build DLL in C, 120
Build DLL in Fortran, 119
Build DLLs, 121
Hello program, 2
Makefile, 52
MPI Hello World, 76
OpenMP Task in C, 285
OpenMP Task in Fortran, 285
prefecth pragma, 109
prefetch directives, 108
SYSTEM_CLOCK use, 46
Vector operation using SSE, 33
Executable
make available, 113
Execution
timing, 45
Extended asm macros, 185

F

F90
aggregate data types, 190
fcon pragma, 330
FFTs library, 125
Filename
conventions, 3
extensions, 4
input files conventions, 4
output file conventions, 5
Files
.def for DLL, 119
case, 279
distributing, 141
licensing redistributable files, 143
names, 3
redistributable, 142
Flags
floating point, 348, 358
MXCSR, 358
register, 341
RFLAGS, 348
Floating point
control word, 348, 358
flags, 341
return values, 341
scratch registers, 341
stack, 227
FLUSH directive, 292
Fortran
Calling C++ Example, 156
data type representation, 187
directive summary, 102, 109
Linux86-64 types, 352
named common blocks, 150
program calling C++ function, 156
types in Win64, 361
Fortran Parallelization Directives
DOACROSS, 289, 289
ORDERED, 293
fPIC, 163, 165
Frame
pointer, 358
Frames
pointer, 341, 344, 347, 350, 360
Function Inlining
inlining and makefiles, 52
inlining examples, 53
inlining restrictions, 53
Functions, 148
calling sequence, 339, 356
Calling sequence, 346
inlining, 52
inlining for optimization, 24
overloaded names, 323
returning scalars, 342, 348, 358
return structures, 342, 348, 358
return unions, 342, 348, 358
return values, 342, 348, 353, 358
stack contents, 342

H

header files
Mac OS X, 12
SUA/SFU, 11
Hello example, 2
Help
on command-line options, 18
parameters, 19
using, 18
HP-MPI
using, 77

I

i8, 163, 165
Information
compiler, 327
Inline assembly
C/C++, 171
clobber list, 176, 176
extended, 172
extended, input operands, 175
extended, output operands, 173
Inlining
automatic, 49
C/C++ restrictions, 54
controls, 263
create inline library, 51
error detection, 53
implement library, 52
invoke function inliner, 49
libraries, 49, 50
Makefiles, 52
-Mextract option, 51
-Minline option, 49
restrictions, 49, 53
specify calling levels, 50
specify library file, 50
suboptions, 49
update libraries, 52
Input
operands, 175
Install
Linux portability package, 142
Instruction
counting, 45
integral
return values, 341
Inter-language Calling, 147, 363
%VAL, 151
arguments and return values, 151
array indices, 152

C$PRAGMA C, 110
C++ calling C, 155
C++ calling Fortran, 157
C calling C++, 155
character case conventions, 149
character return values, 151
compatible data types, 149
Fortran calling C, 153
Fortran calling C++, 156
mechanisms, 148
underscores, 110, 149
Inter-language calling, 353
Interprocedural Analysis, 39
Intrinsics, 171
header file organization, 186
invarif directive, 330
invarif pragma, 330
Invoke
function inliner, 49
IPA, 21, 24
build file location, 43
building without, 39, 40
collection phase, 41
large object file, 43
mangled names, 44
-MIPA issues, 43
multiple-step program, 42
phases, 41
program example, 41
program using Make, 42
propagation phase, 41
recompile phase, 41
single step program, 41
ivdep directive, 330

J

JIT debugging, 139

K

Keywords
C/C++, 194

L

Language options, 258
LAPACK library, 125
Levels
optimization, 44
LIB3F library, 125
libnuma, 46
PGI library, 47
libpgbind, 46
PGI library, 46
Libraries
-Bdynamic option, 121, 203
BLAS, 125
-Bstatic_pgi option, 204
-Bstatic option, 204
C++
standard template, 125
create inline, 51
defined, 111
dynamic, 119
dynamic-link on Windows, 118
dynamic on MAC OS X, 115
FFTs, 125
import, 119
import DLL, 119
inline directory, 52
inlining, 49
LAPACK, 125
LD_LIBRARY_PATH, 113
lib.il, 51
LIB3F, 125
-Mextract option, 51
name, 119
options, 111
portability-related, 46
runtime considerations, Linux, 141
runtime on Windows, 116
run-time routines, 65
SFU/SUA shared object files, 114
shared object files, 112
static, 118
static on Windows, 116
STLPort Standard C++, 125
using inline, 50
Limitations
large array programming; Arrays: limitations, 166
link
static libraries, 204
Linux
64-bit deployment considerations, 142
deploying, 141
header files, 9
large static data, 164
parallelization, 9
portability package install, 142
portability restrictions, 142
redistributable file licensing, 143
redistributable files, 142
use PGI compilers, 9
Listing Files, 275, 279, 279
Loops
failed auto-parallelization, 36
innermost, 36
optimizing, 24
parallelizing, 35
privatization, 37
scalars, 37
timing, 36
unrolling, 24, 29, 333
unrolling, instruction scheduler, 29
unrolling, -Minfo option, 30
ls command, 52
lstval directive, 330, 330
lstval pragma, 330

M

Mac OS X
debug requirements, 12
dynamic libraries, 115
header files, 12
Parallelization, 13
use PGI compilers, 12
MAC OS X
linking, 12
Macros
extended asm, 185
GET_SP, 185
va_arg, 350
Make
IPA program example, 42
utility, 40
Makefiles

example, 52
with options, 18
Mangling
C++ names, 323
function names, 324
operator function names, 324
runttime variable names, 324
static data member names, 324
types, 324
virtual function table variables, 324
Maskedll, 119
MASTER directive, 292
mcmodel=medium, 165
Mlarge_arrays, 165
Mlargeaddressaware, 165
Mmakeimplib, 119
Modifiers
assembly string, 183
characters, 183
MPI
generate profile data, 78
Hello World Example, 76
HP-MPI, 77
implementation options, 74
implementations, 73
Mpich-1 libraries, 75
Mpich-2 libraries, 76
MSMPI, 78
Mvapich libraries, 76
Profile Applications, 75
using, 73
MPI_ROOT, 77
use with HP-MPI, 77
MPICH-1
using, 75
MPICH-2
using, 76
MPIDIR, 76, 76, 77
use with MPICH-1, 76, 76, 77
MPI environment variables
CCP_HOME, 78
CCP_SDK, 78
MPI_ROOT, 77
MPIDIR, 76, 76, 77
MSMPI
using, 78
Multiple systems
-tp option, 21
Multi-Threaded Programs
portability, 46
Mvapich
using, 76
MXCSR register, 358

N

Name mangling
local class, 325
nested class, 325
template class, 325
type, 324
Names
conventions, 352
entities, 323
external, 323
Fortran conventions, 362
mangled name format, 324
mangled runtine variables, 324
mangled static data members, 324
mangled virtual function table variables, 324
NCPUS; Environment variables
NCPUS, 36
noaltcode directive, 328
noaltcode pragma, 328
noassoc directive, 329
noassoc pragma, 329
nobounds directive, 329
nocncall directive, 329
nocncall pragma, 329
noconcur directive, 329
noconcur pragma, 329
nodepchk directive, 329
nodepchk pragma, 329
noeqvchk directive, 329
noeqvchk pragma, 329
nofcon pragma, 330
noinvarif directive, 330
noinvarif pragma, 330
nolstval pragma, 330
NOMIXED_STR_LEN_ARG
ATTRIBUTES directive, 335
nosafe_lastval directive, 331
nosafe_lastval pragma, 331
nosafe pragma, 331
nosafeptr pragma, 332
nosingle pragma, 333
nounroll directive, 333
nounroll pragma, 333
novector directive, 334
novector pragma, 334
novintr directive, 334
novintr pragma, 334

O

OMP_DYNAMIC, 71, 308
omp_get_ancestor_thread_num(), 66
OMP_MAX_ACTIVE_LEVELS, 71, 308
OMP_NESTED, 71, 308
OMP_NUM_THREADS, 71, 309
OMP_SCHEDULE, 71, 309
OMP_STACK_SIZE, 71, 309
OMP_THREAD_LIMIT, 71, 310
OMP_WAIT_POLICY, 310
omp flush pragma, 292
omp for pragma, 290
omp master pragma, 292
omp ordered pragma, 293
omp parallel pragma, 294, 299
omp parallel sections pragma, 296
omp sections pragma, 298
omp threadprivate pragma, 302
on Linux, 141
OpenMP
barrier, 284
environment variables, 308
Fortran Directives, 55
task, 58, 283
task scheduling, 283
taskwait, 284
using, 55
OpenMP C/C++ Pragmas, 55, 283
flush, 292
omp critical, 287
omp master, 292
omp ordered, 293
omp parallel, 299

omp parallel sections, 296
omp sections, 298
omp threadprivate, 302
parallel, 294
parallel sections, 297
OpenMP C/C++ Support Routines
omp_destroy_lock(), 70
omp_get_active_level(), 66
omp_get _ancestor_thread_num(), 66
omp_get_dynamic(), 68
omp_get_max_threads(), 66, 66, 67
omp_get_nested(), 68
omp_get_num_threads(), 65, 65
omp_get_schedule(), 69, 69, 69
omp_get_stack_size(), 67
omp_get_team_size(), 67
omp_get _thread_num(), 66
omp_get_wtick(), 69
omp_in_parallel(), 68
omp_init_lock(), 69
omp_set_dynamic(), 68
omp_set_lock(), 70
omp_set_nested(), 68
omp_set_num_threads(), 66
omp_set_stack_size(), 67
omp_test_lock(), 70
omp_unset_lock(), 70
OpenMP environment variables
MPSTKZ, 132
OMP_DYNAMIC, 71, 130, 130, 308
OMP_MAX_ACTIVE_LEVELS, 71, 308
OMP_NESTED, 71, 130, 308
OMP_NUM_THREADS, 71, 130, 309
OMP_SCHEDULE, 71, 309
OMP_THREAD_LIMIT, 71
OpenMP Fortran Directives, 283, 327
ATOMIC, 286
BARRIER, 287
CRITICAL, 287
DO, 290
FLUSH, 292
MASTER, 292
ORDERED, 293
PARALLEL, 294
PARALLEL DO, 296, 296
PARALLEL SECTIONS, 296
PARALLEL WORKSHARE, 297, 298
SECTIONS, 298
SINGLE, 299
TASK, 300, 301
THREADPRIVATE, 302
WORKSHARE, 303
OpenMP Fortran Support Routines
omp_destroy_lock(), 70
omp_get_ancestor_thread_num(), 66
omp_get_dynamic(), 68
omp_get_level(), 66, 66
omp_get_max_threads(), 66
omp_get_nested(), 68
omp_get_num_procs(), 67
omp_get_num_threads(), 65
omp_get_schedule(), 69, 69
omp_get_stack_size(), 67
omp_get_team_size(), 67
omp_get_thread_num(), 66
omp_get_wtick(), 69
omp_get_wtime(), 69
omp_in_parallel(), 68
omp_init_lock(), 69
omp_set_dynamic(), 68
omp_set_lock(), 70
omp_set_nested(), 68
omp_set_num_threads(), 66
omp_set_stack_size(), 67
omp_test_lock(), 70
omp_unset_lock(), 70
OpenMP pragmas
omp atomic, 286
OpenMP Pragmas
omp for, 290
Operand
aliases, 183
constraints, see constraints, 177
modifier *, 182, 182
modifier &, 182
modifier %, 182
modifier +, 182
modifier =, 182
Operand constraints
machine, 179
Optimization, 23
C/C++ pragmas, 45, 102
C/C++ pragmas scope, 105
cache tiling, 273
default level, 28
default levels, 44
defined, 24
Fortran directives, 45, 101, 327
Fortran directives scope, 104
function inlining, 14, 24, 49
global, 24, 28
global optimization, 28
inline libraries, 50
Inter-Procedural Analysis, 24
IPA, 24
local, 24, 27, 45
loops, 24, 269, 270, 270
loop unrolling, 24, 29
-Munroll, 29
no level specified, 27
none, 27
-O, 226
-O0, 27
-O1, 27
-O2, 27
-O3, 27
-O4, 27
-Olevel, 27
options, 23
parallelization, 35
PFO, 25
PGPROF, 23
pointers, 271
prefetching, 271, 271, 271
profile-feedback (PFO), 44
Profile-Feedback Optimization, 25
profiler, 23
using -Mipa, 39
vectorization, 24, 30
Options
alter effects, 327

cache size, 31
-dryrun, 26
frequently used, 21
Mchkfpstk, 135
-Minfo, 26
-Mneginfo, 26
optimizing code, 23
perfromance-related, 20
prefetch, 32
SSE-related, 31
opt pragma, 330
ORDERED directive, 293

P

Parallalization
code speed, 14
PARALLEL directive, 294
PARALLEL DO directive, 296
Parallelization, 23, 24
auto-parallelization, 35
C++ Pragmas, 59
Directives, 55, 286
directives, 327
Directives, defined, 59
directives format, 59
directives usage, 38
failed auto-parallelization, 36, 279
Mac OS X, 13
-Mconcur=altcode, 35
-Mconcur=cncall, 35
-Mconcur=dist, 35
-Mconcur auto-parallelization, 265
NCPUS environment variable, 36
Pragmas, 55, 286
pragmas, 283
pragmas usage, 38
safe_lastval, 37
user-directed, 224
Parallel Programming
automatic shared-memory, 7
data parallel shared-memory, 7
distributed-memeory, 7
Linux, 9
OpenMP shared-memory, 7
run HPF program, 8
run SMP program, 8
styles, 7
PARALLEL SECTIONS directive, 296
PARALLEL WORKSHARE directive, 297
Parameters
passing in registers, 344, 350, 360
type, 344, 345, 360
type, in C, 351
Performance
-fast, 20
-fastsse, 20
-Mipa, 21
-Mpi=fast, 21
options, 20
overview, 19
PGI_Term
abort value, 136
debug value, 136
signal value, 136
trace value, 136
PGI_TERM
noabort value, 136
nodebug value, 136
nosignal value, 136
notrace value, 136
PGI CDK, 76
PGPROF
overview, 23
profile MPI applications, 75
PGPROF profiler, 23
Platforms
specific considerations, 8
supported, 8
Pointers
%rsp, 347, 347, 358, 358
frame, 341, 344, 350, 360
return values, 341
stack, 341
Portability
Linux, 142
Linux package, 142
multi-threaded programs, 46
Pragmas, 101
altcode, 328
assoc, 329
bounds
bounds pragma, 329
C/C++, 3
clauses, 62
cncall, 329
concur, 329
default scope, 102
defined, 59
depchk, 329
eqvchk, 329
fcon, 330
format, 60
global scope, 102
invarif, 330
loop scope, 102
lstval, 330
noaltcode, 328
noassoc, 329
nobounds
nobounds pragma, 329
nocncall, 329
noconcur, 329
nodepchk, 329
noeqvchk, 329
nofcon, 330
noinvarif, 330
nolstval, 330
nosafe, 331
nosafe_lastval, 331
nosafeptr, 332
nosingle, 333
nounroll, 333
novector, 334
novintr, 334
omp atomic, 286
OpenMP C/C++, 55, 283
opt, 330
optimization, 102
PGI Proprietary, 102
prefetch, 330
prefetch example, 109
prefetch syntax, 109, 334
recognition, 60
routine scope, 102

safe, 331
safe_lastval, 331
safeptr, 332
scope, 102, 105
scope rules, 107
see OpenMP, 297
single, 333
Summary table, 60
summary table, 102
syntax, 102
tp, 333
unroll, 333
vector, 334
vintr, 334
Prefetch, 32
directives, 107, 334
directives example, 108
directives sentinel, 108
directives syntax, 108, 334
-Mprefetch, 271
pargma example, 109
pargma syntax, 109, 334
prefetch directive, 330, 330
prefetch pragma, 330
Preprocessor
cpp, 5
Fortran, 5
Processors
architecture, 144
Profile
generate data, 78
MPI applications, 75
Profiler, 23
PGPROF, 75
Programs
extracting, 53
Propagation
IPA phase, 41
Proprietary environment variables
FORTRAN_OPT, 129, 131
GMON_OUT_PREFIX, 129
MP_BIND, 129
MP_BLIST, 129
MP_SPIN, 129
MP_WARN, 129
MPSTKZ, 129
NCPUS, 129
NCPUS_MAX, 129
NO_STOP_MESSAGE, 129
PGI, 130
PGI_CONTINUE, 130
PGI_OBJSUFFIX, 130
PGI_STACK_USAGE, 130
PGI_TERM, 130
PGI_TERM_DEBUG, 130, 130
STATIC_RANDOM_SEED, 130
TMP, 130
TMPDIR, 130

R

ranlib command, 117
Recompile
IPA phase, 41
Redistributable files
licensing on Linux, 143
Linux, 142
Redistributables
Microsoft Open Tools, 143
PGI directories, 143
REFERENCE
ATTRIBUTES directive, 336
Registers
%rax, 351
allocation, 346, 356
flags, 341
floating point, 341
local, 341
MXCSR, 358
non-volatile, 357
parameter passing, 344, 350, 360
RFLAGS, 348
runtime allocation, 340
scratch, 341, 341
usage, 356
usage conventions, 339
x64 systems, 357
Restrictions
inlining, 53
Return values, 151
character, 151
complex, 152
inegral, 341
none, 342
pointers, 341
types, 353
RFLAGS register, 348
Runtime
environment, 339
libraries on Windows, 116
Linux considerations, 141
Run-time
library routines, 65
Run-time Environment, 339

S

safe_lastval directive, 331
safe_lastval pragma, 331
safe pragma, 331
safeptr pragma, 332
Scalars
alignment, 188, 191
C/C++, 190
Fortran data types, 187
last value, 37
Scopes
C++ classes, 323
directives, 101
pragma rules, 107
pragmas, 102
SECTIONS directive, 298
Set
environment variables, 127
SFU/SUA
use PGI compilers, 11
Shared object files
creating, 112
creating in SFU/SUA, 114
using, 112
using in SFU/SUA, 114
Signals
handlers, 348, 358
SINGLE directive, 299
single pragma, 333
siterc files, 13
SSE
example, 33
scalar code generation, 28
vectorization example, 32

Stacks
alignment, 357
argument order, 349
contents, 342
frame, 340, 357
frames, 346
implementing, 344, 350
passing arguments, 359
pointer, 341, 347, 358
traceback, 139
usage conventions, 347
Static
data in Linux, 164
libraries on Windows, 116
STDCALL
ATTRIBUTES directive, 336
calling conventions, 160
Strings
modifiers, assembly, 184
SUA/SFU
Header Files, 11
header files, 11
Parallelization, 11
shared object files, 114
Subroutines, 148
Symbol
name construction, 159
Syntax
command-line options, 2
pragmas, 102
prefetch directives, 108, 108, 334
prefetch pragmas, 109, 334
System
flags, 341
SYSTEM_CLOCK, 45
usage, 45

T

Table
Fortran Data Type Representation, 187
Fortran Directives, 102, 109
MPI Implementation Options, 74
OpenMP Environment Variables, 70
Real Data Type Ranges, 188
Scalar Type Alignment, 188
TASK directive, 300, 301
Tasks
C example, 285
construct, 284
Fortran example, 285
OpenMP overview, 58, 283
scheduling points, 283
taskwait call, 284
thread-compliant, HP-MPI, 77
THREADPRIVATE directive, 302
Timing
CPU_CLOCK, 45
execution, 45
SYSTEM_CLOCK, 45
TOC file, 52
Tools
PGDBG, xxv
PGPROF, xxv, xxv
tp directive, 333
tp pragma, 333
Types
derived, 354, 364, 364
Fortran, 352
Fortran in Win64, 361

U

Underscores
inter-language calling usage, 149
Unified Binaries
command-line switches, 145, 145
directives, 145
-Mipa option, 39
optimization, 39
-tp option, 39
UNIX
calling conventions, 161
unroll directive, 333
unroll pragma, 333
Use
PGI compiler, 1
User rc files, 13

V

VALUE
ATTRIBUTES directive, 336
vector directive, 334
vector intrinsics
recognition of, 334
Vectorization, 24, 30, 273
associativity conversions, 31
cache size, 31
disable, 334
example using SSE/SSE2, 32
generate packed instructions, 31
generate prefetch instructions, 32
-Mvect, 30
operation control, 31
SSE
option, 31
SSE instructions, 274, 274
sub-options, 30
vector pragma, 334
vintr directive, 334
vintr pragma, 334

W

Win32 Calling Conventions
C, 158, 160
default, 158, 159, 160
STDCALL, 158, 160
symbol name construction, 159
UNIX-style, 158, 160
Windows
command prompt, 11
deploying
Deployment, 143
dynamic-link libraries, 118
runtime libraries, 116
static libraries, 116
use PGI compilers, 10
WORKSHARE directive, 303